is interpreted by the compiler as

```
d2 = d1.operator ++ ();
```

Let us envisage that this operator-overloading function should first increment 'iFeet' portion of 'd1'. It should leave the 'fInches' portion of 'd1' unaltered. Then it should return the resultant object. With these guidelines in mind, the prototype and definition of the operator-overloading function will be as follows.

```
/*Beginning of Distance.h*/
class Distance
{
  public:
    /*
      rest of the class Distance
    */
    Distance operator ++ ();
};
/*End of Distance.h*/

/*Beginning of Distance.cpp*/
#include"Distance.h"
Distance Distance :: operator ++ ()
{
  return Distance(++iFeet, fInches);
}
/*
  definitions of the rest of the functions of class
  Distance
*/
/*End of Distance.h*/
```

Listing 8.12 Declaring member function to overload the increment operator

The operator-overloading function should be `public` because it will mostly be called from within functions that are not members of the class 'Distance'. It should not be a constant member function since it will certainly modify the value of at least one of the data members ('iFeet') of the calling object. Although the definition of the operator-overloading function appears cryptic, it is in fact very simple (and economical). First, the increment operator works (since it is in prefix notation). Thus, the 'iFeet' data member of the calling object gets incremented. Second, the explicit call to the constructor creates a nameless object of the class 'Distance' by passing the incremented value of 'iFeet' and the unaltered value of 'fInches' as parameters. Third, the operator-overloading function returns the nameless object thus constructed. If the call to the operator-overloading function

is on the right-hand side of the assignment operator, the values of the returned object will expectedly be copied to the object on the left. Thus, our purpose is served.

However, we would like a different effect to be produced if we write the statement

```
d2 = d1++;
```

In this case, we would like the initial value of 'd1' to be copied to 'd2' and, thereafter, the value of 'iFeet' data member of 'd1' to get incremented. However, if the compiler interprets both the statements

```
d2 = ++d1;
```

and

```
d2 = d1++;
```

in identical ways, then we will have no way of writing the two different functions. Fortunately, this is not so. While the compiler interprets the statement

```
d2 = ++d1;
```

as

```
d2 = d1.operator ++ ();
```

it interprets the statement

```
d2 = d1++;
```

as

```
d2 = d1.operator ++ (0);
```

It implicitly passes zero as a parameter to the call to the operator-overloading function when the postfix notation is used. If it finds a prototype that matches this call exactly, it compiles without warnings or errors. However, if it finds the prototype given in Listing 8.12, it gives a warning but still compiles with the operator-overloading function 'Distance :: operator ++ ()'. The fact that the compiler first looks for a function with an integer as a formal argument provides us with a solution. We can now define an additional operator-overloading function to overload the increment operator in postfix notation.

```
/*Beginning of Distance.h*/
class Distance
{
  public:
    Distance operator ++ ();        //for prefix notation
    Distance operator ++ (int);     //for postfix notation
    /*
      rest of the class Distance
    */
};
/*End of Distance.h*/

/*Beginning of Distance.cpp*/
#include"Distance.h"
Distance Distance :: operator ++ ()      //for prefix
                                         //notation
{
  return Distance(++iFeet, fInches);     //as in listing
                                         //8.12
}

Distance Distance :: operator ++ (int)   //for postfix
                                         //notation
{
  return Distance(iFeet++, fInches);
}

/*
  definitions of the rest of the functions of class
  Distance
*/
/*End of Distance.cpp*/
```

Listing 8.13 Overloading the 'increment' operator in both the prefix as well as the postfix notation

The explanation for the definition of the function to overload the 'increment' operator in postfix notation is as follows. The constructor gets called before the 'increment' operator executes because the 'increment' operator has been purposefully placed in postfix notation. Thus, a nameless object with the initial values of the calling object is created. Thereafter, the 'increment' operator increments the value of 'iFeet' data member of the calling object. Finally, the nameless object constructed earlier with the initial values of the calling object is returned. Since the formal parameter of the function is a dummy, therefore, its name need not be mentioned. Obviously, if the call to this operator-overloading function is on the right-hand side of the 'assignment' operator and there is an object of the class 'Distance' on its left, then the object on the left will get the initial values of the object on

the right. The value of the object on the right will alone be incremented. These two operator-overloading functions convincingly duplicate the default action of the 'increment' operator on intrinsic types.

Obviously, if we provide an operator-overloading function for the 'increment' operator in prefix notation, we must provide one for the postfix notation also.

Decrement operators are overloaded in the same way as the increment operators.

```
/*Beginning of Distance.h*/
class Distance
{
  public:
    Distance operator ++ ();
    Distance operator ++ (int);
    Distance operator -- ();
    Distance operator -- (int);
    /*
      rest of the class Distance
    */
};
/*End of Distance.h*/

/*Beginning of Distance.cpp*/
#include"Distance.h"
Distance Distance :: operator -- ()
{
  return Distance(--iFeet, fInches);
}

Distance Distance :: operator -- (int)
{
  return Distance(iFeet--, fInches);
}

/*
  definitions of the rest of the functions of class
  Distance
*/
/*End of Distance.cpp*/
```

Listing 8.14 Overloading the decrement operator in both the prefix and postfix notation

Overloading the Unary Minus and the Unary Plus Operator

Overloading the 'unary minus' operator is shown in the following listing.

```
/*Beginning of A.h*/
class A
{
    int x;
  public:
    A(int = 0);
    A operator - ();
};
/*End of A.h*/
/*Beginning of A.cpp*/
#include"A.h"
A A : :operator - ()
{
  return A(-x);
}
/*End of A.cpp*/
```

Listing 8.15 Overloading the 'unary minus' operator through a member function

The operator can be overloaded by a friend function also (as in Listing 8.16).

```
/*Beginning of A.h*/
class A
{
    int x;
    A(int = 0);
  public:
    friend A operator - (const A&);
};
/*End of A.h*/
/*Beginning of A.cpp*/
#include"A.h"
A operator - (const A & AObj)
{
  return A(-AObj.x);
}
/*End of A.cpp*/
```

Listing 8.16 Overloading the 'unary minus' operator through a friend function

Overloading the 'unary plus' operator is left as an exercise for the reader.

Overloading the Arithmetic Operators

Arithmetic operators are binary operators. Therefore, the syntax for overloading them through member functions is as follows:

```
class <class_name>
{
  public:
    //prototype
    <return_type> operator<arith_op_symbol>(<param_list>);
};

//definition
<return_type> <class_name>::operator<arith_op_symbol>
                              (<param_list>)
{
  //function body
}
```

Listing 8.17 Syntax for overloading the arithmetic operators through member functions

An object that will store the value of the right-hand side operand of the arithmetic operator will appear in the list of formal arguments. The left-hand side operand will be passed implicitly to the function since the operator-overloading function will be called with respect to it. The statement

```
Obj3 = Obj1 <arith_op_symbol> Obj2;
```

will be interpreted as

```
Obj3 = Obj1.operator <arith_op_symbol> (Obj2);
```

If instead a friend function overloads the arithmetic operator, the syntax will be as follows:

```
class <class_name>
{
  public:
    //prototype
    friend <return_type>operator<arith_op_symbol>
                          (<param_list>);
};
```

```
//definition
<return_type> operator<arith_op_symbol>(<param_list>)
{
  //function body
}
```

Listing 8.18 Syntax for overloading the arithmetic operators through friend functions

Objects that store the values of the left-hand side and the right-hand side operands of the arithmetic operator will appear in the list of formal arguments.

The statement

```
Obj3 = Obj1 <arith_op_symbol> Obj2;
```

will be first interpreted as

```
Obj3 = Obj1.operator <arith_op_symbol> (Obj2);
```

Since, the arithmetic operator has been overloaded through a friend function, the final interpretation will be

```
Obj3 = operator <arith_op_symbol> (Obj1,Obj2);
```

Now let us try some concrete examples. Let us find out how to overload the 'addition' operator for the class 'Distance' with which we are already familiar. We would like the following piece of code to compile successfully and its output to be 10'-2'.

```
Distance d1(5,8),d2(4,6),d3;
d3=d1+d2;
cout<<d3.getFeet()<<"'-"<<d3.getInches()<<"''\n";
```

Listing 8.19 Using an overloaded 'addition' operator on objects of the class 'Distance'

For this, we must overload the 'addition' operator for the class 'Distance' by using either a member function or a friend function.

Let us first look at a member function to overload the 'addition' operator for the class 'Distance'. In this case, the statement

```
d3 = d1 + d2;
```

will be interpreted as

```
d3 = d1.operator + (d2);
```

This obviously means that the function must return an object of the class 'Distance' and must accept an object of the class 'Distance' as a parameter. The actual code to implement the 'addition' operator so that it produces the desired effect described above is as follows.

```
/*Beginning of Distance.h*/
class Distance
{
    int iFeet;
    float fInches;
  public:
    Distance(const int=0, const float=0.0);
    void setFeet(const int=0);
    int getFeet() const;
    void setInches(const float=0.0);
    float getInches() const;

    //prototype
    Distance operator + (const Distance) const;
};
/*End of Distance.h*/

/*Beginning of Distance.cpp*/
#include"Distance.h"
//definition
Distance Distance::operator+(const Distance dd1) const
{
  return Distance(iFeet+dd1.iFeet, fInches+dd1.fInches);
}

/*
  definitions of the rest of the functions of class
  Distance
*/
/*End of Distance.cpp*/
```

Listing 8.20 Overloading the 'addition' operator for the class 'Distance' through member function

The code in Listing 8.20 works fine if the right-hand side operand of the 'addition' operator is an object of class 'Distance'. However, if it is a float type value, then the preceding function will not work.

```
d3=d1+4.5;
```

This is because the compiler will interpret this statement as follows:

```
d3=d1.operator+(4.5);
```

The float type value 4.5 will be passed as a parameter to the operator-overloading function. Since the formal argument of the operator-overloading function is a 'Distance' type object, the compiler will throw an error. However, introducing a suitable constructor that converts from float to 'Distance' solves the problem.

```
/*Beginning of Distance.h*/
class Distance
{
  public:
    Distance(const float);
    /*
      rest of the class Distance
    */
};
/*End of Distance.h*/

/*Beginning of Distance.cpp*/
#include"Distance.h"
Distance::Distance(const float p)
{
  iFeet=(int)p;
  fInches=(p-iFeet)*12;
}

/*
  definitions of the rest of the functions of class
  Distance
*/
/*End of Distance.cpp*/
```

Listing 8.21 Introducing a constructor in the class 'Distance' to initialize its objects to float type values

However, one condition still remains to be tackled. What if the left-hand side operand is of float type?

```
d2 = 4.75 + d1;
```

The solution is obvious. We replace the member function given in Listing 8.20 with a friend function.

```
/*Beginning of Distance.h*/
class Distance
{
  public:
    //no 'Distance operator + (const Distance) const;'

    //prototype
    friend Distance operator + (const Distance , const
                                Distance);
    /*
      rest of the class Distance
    */
};
/*End of Distance.h*/

/*Beginning of Distance.cpp*/
#include"Distance.h"
//definition
Distance operator + (const Distance dd1, const Distance
                     dd2)
{
  return Distance(dd1.iFeet+dd2.iFeet,
                  dd1.fInches+dd2.fInches);
}

/*
  definitions of the rest of the functions of class
  Distance
*/
/*End of Distance.cpp*/
```

Listing 8.22 Overloading the 'addition' operator for the class 'Distance' through friend function

The friend function given in Listing 8.22 tackles all three conditions as follows:

- Both left-hand side and right-hand side operands are objects of class 'Distance':

  ```
  d3 = d1 + d2;
  ```

  The operator-overloading function is called straight away without any prior conversions.

- Right-hand side operand is a float type value while left-hand side operand is an object of class 'Distance':

  ```
  d2 = d1 + 4.75;
  ```

The right-hand side operand is first converted into an object of the class 'Distance' by the constructor and then the operator-overloading function is called.

- Left-hand side operand is a float type value while right-hand side operand is an object of class 'Distance':

```
d2 = 4.75 + d1;
```

The left-hand side operand is first converted into an object of the class 'Distance' by the constructor and then the operator-overloading function is called.

We may wonder about the fourth possibility where both operands are float type values. However, in that case the operator-overloading mechanism will not be invoked at all. Instead, the float type values will simply get added to each other.

The statement

```
d1 = 4.75 + 3.25;
```

will turn into

```
d1 = 8.0;
```

However, there is no function in the class 'Distance' that converts a float type value to an object of class 'Distance'. Surprisingly, in this case also, the constructor that takes a float type value as a parameter and initializes the object with it will be called. This is despite the fact that the object is being created and initialized by two separate statements. Such a constructor is called an implicit constructor.

Note that in Listing 8.22, the member function to overload the 'addition' operator is replaced by a friend function. Having both a friend function and a member function will lead to ambiguity errors.

The compiler will be able to resolve the call

```
d3 = d1 + d2;
```

by both

```
//member function
Distance Distance::operator + (const Distance);
```

and

```
//friend function
Distance operator + (const Distance, const Distance);
```

This will naturally confuse the compiler.

We have now reached the end of our discussion on overloading the 'addition' operator. The method of overloading the remaining arithmetic operators is left as an exercise for the reader.

Overloading The Relational Operators

Relational operators are binary operators. Therefore, the syntax for overloading them through member functions is as follows:

```
class <class_name>
{
  public:
    //prototype
    <return_type> operator <rel_op_symbol> (<param_list>);
};

//definition
<return_type> <class_name>::operator <rel_op_symbol>
                            (<param_list>)
{
  //function body
}
```

Listing 8.23 Syntax for overloading the relational operators through member functions

An object that will store the value of the right-hand side operand of the relational operator will appear in the list of formal arguments. The left-hand side operand will be passed implicitly to the function since the operator-overloading function will be called with respect to it. The expression

```
Obj1 <rel_op_symbol> Obj2
```

will be interpreted as

```
Obj1.operator <rel_op_symbol> (Obj2)
```

If instead, a friend function overloads the relational operator, the syntax will be as follows:

```
class <class_name>
{
  public:
    //prototype
    friend <return_type> operator <rel_op_symbol>
                          (<param_list>);
};
//definition
<return_type> operator <rel_op_symbol> (<param_list>)
{
  //function body
}
```

Listing 8.24 Syntax for overloading the relational operators through friend functions

Objects that store the values of both the left-hand side and the right-hand side operands of the relational operator will appear in the list of formal arguments.

The expression

```
Obj1 <rel_op_symbol> Obj2
```

will first be interpreted as

```
Obj1.operator <rel_op_symbol> (Obj2)
```

Since, the relational operator has been overloaded through a friend function, this interpretation will be

```
operator <rel_op_symbol> (Obj1,Obj2)
```

Now let us find out how to overload the 'greater than' relational operator for the class 'Distance'. We would like the following piece of code to compile successfully and its output to be "Greater than".

```
Distance d1(5,8),d2(4,6);
if(d1>d2)
  cout<<"Greater than";
else
  cout<<"Less than";
```

Listing 8.25 Using an overloaded 'greater than' operator for the class 'Distance'

For this, we must overload the 'greater than' operator for the class 'Distance' by using either a member function or a friend function.

Let us first look at a member function to overload the 'greater than' operator for the class 'Distance'. In this case, the expression

```
d1>d2
```

will be interpreted as

```
d1.operator>(d2)
```

Obviously, the function must return a boolean type value (true or false) and should accept an object of the class 'Distance' as a parameter. The actual code to implement the 'greater than' operator so that it produces the desired aforementioned effect is as follows.

```
/*Beginning of Distance.h*/
enum bool{false, true};
class Distance
{
    int iFeet;
    float fInches;
  public:
    Distance(const int=0, const float=0.0);
    bool operator > (const Distance) const;  //prototype
    /*
      rest of the class Distance
    */
};
/*End of Distance.h*/

/*Beginning of Distance.cpp*/
#include"Distance.h"
//definition
bool Distance::operator > (const Distance dd1) const
{
  if(iFeet*12+ fInches >dd1.iFeet*12 +dd1.fInches)
    return true;
  return false;
}
/*
  definitions of the rest of the functions of class
  Distance
*/
/*End of Distance.cpp*/
```

Listing 8.26 Overloading the 'greater than' operator for the class 'Distance' through a member function

The code in Listing 8.26 works fine if the right-hand side operand of the 'greater–than' operator is an object of class 'Distance'. However, if it is a float type value, then the expression will not compile.

```
d1>4.5
```

This is because the compiler will interpret this expression as follows:

```
d1.operator>(4.5);
```

The float type value '4.5' will be passed as a parameter to the operator-overloading function. Since the formal argument of the operator-overloading function is a 'Distance' type object, the compiler will throw an error. As in the case of the 'addition' operator, introducing a suitable constructor that converts from float to 'Distance' solves the problem (see Listing 8.21).

Nevertheless, one condition still remains to be tackled. What will happen if the left-hand side operand is of float type?

```
4.75 > d1
```

The solution is the same as in the case of the 'addition' operator. We replace the member function given in Listing 8.26 with a friend function.

```
/*Beginning of Distance.h*/
enum bool {false, true};
class Distance
{
  public:
    //no 'bool operator > (const Distance) const;'
    //prototype
    friend bool operator > (const Distance , const
                            Distance);
    /*
      rest of the class Distance
    */
};
/*End of Distance.h*/

/*Beginning of Distance.cpp*/
#include"Distance.h"
//definition
bool operator > (const Distance dd1, const Distance dd2)
```

```
{
  if(dd1.iFeet*12+ dd1.fInches >dd2.iFeet*12+dd2.fInches)
    return true;
  return false;
}

/*
  definitions of the rest of the functions of class
  Distance
*/
/*End of Distance.cpp*/
```

Listing 8.27 Overloading the 'greater-than' operator for the class 'Distance' through friend function

The friend function given in Listing 8.27 tackles all three conditions as follows:

- Both left-hand side and right-hand side operands are objects of class 'Distance':

  ```
  d1 > d2
  ```

  The operator-overloading function is called straight away without any prior conversions.

- Right-hand side operand is a float type value while left-hand side operand is an object of class 'Distance':

  ```
  d1 > 4.75
  ```

  The right-hand side operand is first converted into an object of the class 'Distance' by the constructor and then the operator-overloading function is called.

- Left-hand side operand is a float type value while right-hand side operand is an object of class 'Distance':

  ```
  4.75 > d1
  ```

  The left-hand side operand is first converted into an object of the class 'Distance' by the constructor and then the operator-overloading function is called.

We may again wonder about the fourth possibility where both operands are float type values. Again, in such a case the operator-overloading mechanism will not be invoked at all. Instead, the float type values will simply get compared to each other.

The expression

```
4.75 > 3.25
```

will return true.

As in the case of the 'addition' operator, the member function to overload the 'greater than' operator is replaced by a friend function. Having both a friend function and a member function will lead to ambiguity errors.

The compiler will be able to resolve the expression

```
d1 > d2
```

by both

```
//member function
bool Distance::operator > (const Distance);
```

and

```
//friend function
bool operator > (const Distance, const Distance);
```

This will naturally confuse the compiler.

We have now reached the end of our discussion on overloading the 'greater than' operator. The method of overloading the remaining relational operators is left as an exercise for the reader.

Overloading the Assignment Operator

The 'assignment' operator is a binary operator. If overloaded, it must be overloaded by a non-static member function only. Thus, the syntax for overloading the 'assignment' operator is

---

```
class <class_name>
{
  public:
    //prototype
    class_name & operator = (const class_name &);
};
```

```
class_name & class_name :: operator = (const class_name &
rhs) //definition
{
  //statements
}
```

Listing 8.28 Syntax for overloading the 'assignment' operator

We must keep in mind that, by default, the compiler generates the function to overload the 'assignment' operator if the class designer does not provide one. This default function carries out a simple member-wise copy.

```
class A
{
  public:
    A& operator = (const A&);
};

A& A :: operator = (const A& rhs)
{
  *this = rhs;
  return *this;
}
```

Listing 8.29 Default 'assignment' operator generated by the compiler

In most cases, this default 'assignment' operator is sufficient. However, there are cases where this default behavior causes problems. We may recollect the section on copy constructors from Chapter 4. We discussed the ill effects of the default copy constructor for classes that acquire resources dynamically. Exactly the same problems arise due to the effect of the default 'assignment' operator. The problems caused by the code

```
String s1, s2;
s1.setString("abcd");
s2 = s1;
```

if the 'assignment' operator is not defined are the same as the problems that arise out of the code

```
String s1("abcd");
String s2 = s1;
```

if the copy constructor is not defined. As a result of the preceding 'assignment' operation, the pointers of both 's1' and 's2' will end up pointing at the same memory block

(see Diagram 8.1). From the study of the copy constructor, we are already conversant with the havoc this situation causes. The conclusion is that the 'assignment' operator must be defined for a class for whom the copy constructor has been defined. A suitable definition of the 'assignment' operator for the class 'String' follows.

```
/*Beginning of String.h*/
class String
{
    char * cStr;
    unsigned int len;
  public:
    String(const String&);     //the copy constructor
    String& operator = (const String&);
    /*
      rest of the class String
    */
};
/*End of String.h*/

/*Beginning of String.cpp*/
#include"String.h"
String& String :: operator = (const String& ss)
{
  if(this != &ss)
  {
    if(cStr != NULL)
    {
      delete[] cStr;
      cStr = NULL;
      len = 0;
    }
    if(ss.cStr != NULL)
    {
      len = ss.len;
      cstr = new char[len + 1];
      strcpy(cStr,ss.cStr);
    }
  }
  return *this;
}

/*
  definitions of the rest of the functions of class String
*/
/*End of String.cpp*/
```

Listing 8.30 A practical example of overloading the 'assignment' operator

Before understanding why the outermost 'if' (`if(this != &ss)`) has been inserted at the top of the function and why the function returns the calling object by reference, we must appreciate that the definition of the 'assignment' operator in Listing 8.30 convincingly handles all four possible cases as follows:

- **`LHS.cStr`** `= NULL and` **`RHS.cStr`** `= NULL`

  If **`LHS.cStr`** `= NULL` then the first inner 'if' (`if(cStr != NULL)`) fails and the corresponding 'if' block does not execute. If **`RHS.cStr`** `= NULL` then the second inner 'if' (`if(ss.cStr != 0)`) fails and the corresponding 'if' block does not execute. The entire function as a whole does not do anything except that it returns the calling object by reference. As expected and desired, the value of the left-hand side operand remains unchanged (as **`cStr`** `= NULL` and **`len`** `= 0`) because the corresponding values in the right-hand side object are NULL and 0, respectively.

- **`LHS.cStr`** `= NULL and` **`RHS.cStr`** `!= NULL`

  If **`LHS.cStr`** `= NULL` then the first inner 'if' (`if(cStr != NULL)`) fails and the corresponding 'if' block does not execute. If **`RHS.cStr`** `!= NULL` then the second inner 'if' (`if(ss.cStr != 0)`) succeeds and the corresponding 'if' block executes. It does the following:

  - correctly sets the value of the 'len' member of the calling object to be equal to the length of the memory block that will hold a copy of the string at which 'cStr' member of the right-hand side object is pointing,
  - allocates just enough memory to hold a copy of the string at which the **cStr** member of the right-hand side object is pointing and makes the 'cStr' member of the left-hand side object point at it, and
  - copies the string at which the 'cStr' member of the right-hand side object is pointing into the memory block at which the 'cStr' member of the left-hand side object is pointing.

- **`LHS.cStr`** `!= NULL and` **`RHS.cStr`** `= NULL`

  If **`LHS.cStr`** `!= NULL` then the first inner 'if' (`if(cStr != NULL)`) succeeds and the corresponding 'if' block executes. It deallocates the memory block at which the 'cStr' member of the left-hand side object points, sets its value to NULL and sets the value of 'len' member of the left-hand side object to 0. If **`RHS.cStr`** `= NULL` then the second inner 'if' (`if(ss.cStr != 0)`) fails and the corresponding 'if' block does not execute. As expected and desired, if it was not already so, the value of the left-hand side operand gets nullified (**`cStr`** `= NULL` and **`len`** `= 0`) because the right-hand side operand is NULL.

- **LHS.cStr** != NULL and **RHS.cStr** != NULL

  If **LHS.cStr** != NULL then the first inner 'if' (if(**cStr** != NULL)) succeeds and the corresponding 'if' block executes. It deallocates the memory block at which the 'cStr' member of the left-hand side object points, sets its value to NULL, and sets the value of 'len' member of the left-hand side object to 0. If **RHS.cStr** != NULL then the second inner 'if' (if(**ss.cStr** != 0)) succeeds and the corresponding 'if' block executes. It does the following:

  - correctly sets the value of the 'len' member of the calling object to be equal to the length of the memory block that will hold a copy of the string at which 'cStr' member of the right-hand side object is pointing,
  - allocates just enough memory to hold a copy of the string at which the 'cStr' member of the right-hand side object is pointing and makes the 'cStr' member of the left-hand side object point at it, and
  - copies the string at which the 'cStr' member of the right-hand side object is pointing into the memory block at which the 'cStr' member of the left-hand side object is pointing.

Now let us understand why the preceding function to overload the 'assignment' operator accepts the argument as a const reference and also returns the calling object by reference. The function accepts the argument as a const reference to test for and guard against self-assignment. First, let us understand how this guard works. We shall then find out why this check is needed at all.

We must take note of the following two facts:

- Since the formal argument 'ss' in the above function is a reference variable, its address is the same as the address of the right-hand side object.
- The this pointer holds the address of the left-hand side object.

Therefore, the 'if' condition 'this == &**ss**' (address of left-hand side object == address of right-hand side object) tests to find out whether an object is being equated with itself or not. An object may get equated with itself in a variety of ways:

```
String s1;
s1 = s1;
```

or

```
String s1;
String &s2 = s1;
s2 = s1;
```

Each of these assignments will cause an execution of the function to overload the 'assignment' operator. Moreover, in each of the cases, the 'if' condition in that function will evaluate to true. For such circumstances, the main body of the operator-overloading function has been deliberately designed to remain unexecuted. Why is this necessary? The reason is simple—in case of a self-assignment, no action is necessary! This function to overload the assignment will work even if the outer 'if' condition is removed and the reference variable that appears as the formal argument is replaced by an ordinary variable.

```
/*Beginning of String.h*/
class String
{
    char * cStr;
    unsigned int len;
  public:
    String(const String&);     //the copy constructor
    String operator = (const String);
    /*
      rest of the class String
    */
};
/*End of String.h*/

/*Beginning of String.cpp*/
#include"String.h"
String String :: operator = (const String ss)
{
  if(cStr != NULL)
  {
    delete[] cStr;
    cStr = NULL;
    len = 0;
  }
  if(ss.cStr != 0)
  {
    len = ss.len;
    cstr = new char[len + 1];
    strcpy(cStr,ss.cStr);
  }
  return *this;
}

/*
  definitions of the rest of the functions of class String
*/
/*End of String.cpp*/
```

Listing 8.31 Bypassing the check for self-assignment in the function to overload the 'assignment' operator

However, this function proves to be highly inefficient in case of a self-assignment. Suppose the statement is:

```
s1 = s1;
```

This statement turns into

```
s1.operator=(s1);
```

's1' is passed by value to the operator-overloading function. Therefore, the copy constructor is called with respect to the formal arguments 'ss' and 's1' is passed as a parameter to it. A properly defined copy constructor ensures that 'ss' contains a separate copy of the same string which 's1' contains. Nevertheless, the copy constructor is called. Now when the actual function body executes, the string contained by 's1' is first deallocated by the first 'if' block and then reallocated with the same value for the string by the second 'if' block. Although the net effect is that nothing happens to the actual value of the string contained by the object, the function is nevertheless inefficient. The unnecessary deallocation and reallocation can and should be avoided. This has been done by the check for self-assignment given in Listing 8.30.

Next, let us understand why the function has been designed to return by reference. The reasons are similar to those that prompted us to pass by reference (to check for self-assignment). The function has been designed to return by reference to prevent chaining operation from becoming inefficient, that is, to ensure an efficient execution of statements such as the following ones.

```
String s1, s2, s3;
s3 = s2 = s1;
```

This statement is interpreted as

```
s3.operator = (s2.operator = (s1));
```

Suppose the statement is written as

```
s2 = s2 = s1;
```

Notice the self-assignment embedded in the preceding statement. First, 's2' is equated with 's1'. Then the value of 's2' is returned. Suppose it is returned by value and not by reference. In this case, a copy of 's2' is created in the stack. Although the copy has a separate copy of the same string value as 's2' has, its address is nevertheless different from that of 's2'. Therefore, when the 'assignment' operator executes for the second time, the reference variable 'ss' refers to this copy and not to 's2' itself. Consequently,

the test for self-assignment fails and again the unnecessary deallocation and reallocation operations occur.

There is another circumstance when the library programmer would like to overload the 'assignment' operator. The library programmer may not want two objects to share even different copies of the same data. In the previous example, where the 'assignment' operator has been overloaded for the class 'String', objects are able to share physically separate and different copies of the same string value. To satisfy the new requirement described earlier, the 'assignment' operator should be defined as a private member function.

```
class A
{
   A& operator = (const A&);
  public:
    /*
      rest of the class A
    */
};
```

Listing 8.32 Overloading the 'assignment' operator through a private member function

Now, if the client programs call the 'assignment' operator indirectly (`object1 = object2`) or directly (`object1.operator=(object2)`), the compiler raises an error and the assignment of one object to another is prevented. What will happen if one of the member functions or friend functions of class A calls the 'assignment' operator? This compiler will certainly not complain and our safeguard will fail. For this, the library programmer can simply avoid defining the 'assignment' operator. Now, if one of the member functions or friend functions of class A calls the 'assignment' operator, the compiler does not complain, but the linker certainly does!

Let us understand another interesting thing about the 'assignment' operator. For this, we should remember that a derived class object can be assigned to a base class object. However, the reverse is not true. The reason is obvious. Suppose A is the base class and B is its derived class.

```
A A1;
B B1;
A1=B1;      //OK
B1=A1;      //ERROR!
```

Listing 8.33 Assigning a derived class object to a base class object and vice-versa

The set of data members of the derived class is, or is reckoned to be, a proper superset of the set of data members of its base class. Thus, in the example in Listing 8.33, 'B1' will have its own copies of not only those data members that 'A1' has, but also some extra data members of its own. If the second assignment in Listing 8.33 works, then the data members of 'B1' that are common in name with those of 'A1' will get initialized. However, the data members that are exclusively in 'B1' will remain unchanged and may no longer match with rest of the data members of 'B1'. Keeping this in mind, the compiler prevents the second assignment.

However, the class designer, if he/she so desires, may provide an 'assignment' operator function to the derived class so that a base class object can be assigned to a derived class object.

```
/*Beginning of B.h*/
#include"A.h"
class B : public A
{
  public:
    B& operator=(const A&);    //to enable B1=A1;
    /*
      rest of the class B
    */
};
/*End of B.h*/
```

Listing 8.34 Enabling a base class object to be assigned to a derived class object

Statements to modify the values of the data members that are exclusive to the derived class can be provided in Listing 8.34.

Suppose there is no explicitly defined 'assignment operator-overloading' function for the derived class that has a reference to the *derived* class object as a formal argument. Further, suppose there *is* an explicitly defined 'assignment operator-overloading' function for the derived class that has a reference to the *base* class object as a formal argument. Even then the complier would generate an 'assignment' operator that has a reference to the *derived* class object as a formal argument. For suppressing the generation of the implicit default assignment, the formal argument of the explicit operator must be of the *same* type as the class itself

Overloading the Insertion and the Extraction Operators

The syntax for overloading the 'insertion' operator is as follows:

```
class A
{
  public:
    //prototype
    friend ostream & operator << (ostream &, const A &);
    /*
      rest of the class A
    */
};

//definition
ostream & operator << (ostream & dout, const A & AA)
{
  /*
    rest of the function
  */
  return dout;
}
```

Listing 8.35 Syntax for overloading the 'insertion' operator

The statement

```
cout << A1;     //A1 is an object of class A
```

is interpreted as

```
operator << (cout, A1);
```

The syntax for overloading the 'extraction' operator is as follows:

```
class A
{
  public:
    /*
      rest of the class A
    */
    //prototype
    friend istream & operator >> (istream &, A &);
};

//definition
istream & operator >> (istream & din, A & AA)
```

```
{
  /*
    rest of the function
  */
  return din;
}
```

Listing 8.36 Syntax for overloading the 'extraction' operator

The statement

```
cin >> A1;      //A1 is an object of class A
```

is interpreted as

```
operator >> (cin, A1);
```

The 'insertion' and the 'extraction' operators are overloaded by using friend functions for reasons explained in the beginning of this chapter.

We may observe that the objects of the classes 'istream' and 'ostream' are passed and returned by reference in the preceding functions. Let us understand why. The copy constructor and the 'assignment' operator have been declared as protected members in both the classes 'istream' and 'ostream'. This prevents two objects from undesirably sharing even different copies of the same stream. Thus the statements

```
ostream dout = cout;      //ERROR!

ostream dout;
dout = cout;              //ERROR!

istream din = cin;        //ERROR!

istream din;
din = cin;                //ERROR!
```

will throw compile-time errors. This explains why the formal arguments are reference variables.

The compulsion to return by reference is also explained similarly. If the object is returned by value, then a separate object is created in the stack. A call to the copy constructor is dispatched with respect to it and the object returned by the operator-overloading function is passed as a parameter. However, the copy constructor is a protected member! Therefore, the object must be returned by reference and not by value. But why should the object be returned at all. Can the function not return anything? Can the function not be as follows?

```
class A
{
  public:
    //prototype
    friend void operator << (ostream &, const A &);
    /*
      rest of the class A
    */
};
//definition
void operator << (ostream & dout, const A & AA)
{
  /*
    definition of the function
  */
}
```

Listing 8.37 Overloading the 'insertion' operator without returning

The answer is yes. Nevertheless, how will we chain the operator?

```
cout << A1 << A2;
```

The preceding statement is interpreted as

```
operator << (operator << (cout, A1), A2);
```

If the inner nested call (whose return value becomes the first argument of the outer one) returns 'void' instead of 'ostream &', how will the outer call execute?

The 'insertion' and 'extraction' operators are overloaded to achieve data abstraction—a complete independence between the interface and the implementation. The signatures and return types of member functions do not change even if the data members within the class do. Consequently, changes in the internal representation of data members of a class do not force its client programs to change. Client programs need not be aware either of the internal representation of data inside the class whose objects they are operating or of any changes therein.

Let us understand this with the help of an example. Let us consider the class 'String' and the ''String::getString()' function. The function returns a `char *`. This is because the string is stored in a null terminated memory block of characters. Suppose this manner of representing the data changes for some reason. Maybe the string is no longer stored as a null terminated string. Maybe the string is stored in wide characters. These changes necessitate modifications in the client programs. For example, the following statement will no longer work:

```
cout<<s1.getString()<<endl;
```

The problem with having a function return the value to be inserted is that the function must return a value and that value must have a data type—the data type of the data member that is containing the object's value. That is where the problem lies. If the data type changes, the existing clients are likely to fail.

However, if the 'insertion' operator has been overloaded then the preceding statement can be rewritten as:

```
cout<<s1<<endl;
```

The responsibility of displaying the data is shifted to the object itself. The manner in which the data is stored in the object and any change therein will no longer affect the client.

Nevertheless, it seems that even if the client programs need not recompile, their object files will have to be re-linked to the new libraries, which are created out of the changed class definition, to create updated executables. However, in the actual programming world, libraries are provided as Dynamic Link Libraries (DLL). They do not form a part of the executables physically. They exist separately. Whenever the corresponding executable executes, they are dynamically loaded into the memory during run time and if the called functions are contained within them, they are executed. Operator overloading, together with DLLs, enables a library programmer in achieving complete data abstraction.

Can the same effect can be achieved by a friend function that has the same signature as the 'insertion' operator?

```
class String
{
  public:
    friend ostream& print(ostream&, const String&);
    /*
      rest of the class String
    */
};
```

Listing 8.38 A friend function as an alternative to operator overloading

Let us output one object of the class 'String' by using Listing 8.38

```
print(cout,s1);
```

In case of two objects:

```
print(print(cout,s1),s2);
```

In case of three objects:

```
print(print(print(cout,s1),s2),s3);
```

However, the statement

```
cout<<s1<<s2<<s3;
```

looks far more intuitive.

Moreover, what will happen in case of templates? Let us consider a common global template function that has calls to the 'insertion' operator embedded within it. If we want to utilize such a function by passing an object of the class 'String' as a parameter, the 'insertion' operator would automatically get applied on the passed object. If the 'insertion' operator has not been overloaded for the class 'String', compile-time error will arise. The narration and examples on function templates, from the chapter on templates (Chapter 9), clarify this point.

The 'extraction' operators are overloaded for similar reasons as the 'insertion' operators. The problem with 'String::setString()' function is that the client needs to load the string that it wants to store in an object of the class 'String' in a buffer and then pass it to a call to this function. The formal argument of this function is of the same type as the data member that is storing the string. Obviously, the buffer should also be of the same type. The problem with this function is that the buffer must be passed to the function and that array must have the same type as the data member in which the data is stored. If that type changes, the type of the buffer also needs to change. This forces the clients to change. But, if the 'extraction' operator is overloaded for the class 'String', the following statement can be used instead of calls to the 'String::setString()' function:

```
cin>>s1;
```

The responsibility of reading the data is shifted to the object itself. Again, the manner in which the data is stored in the object and any change therein will no longer affect the client.

The 'insertion' and 'extraction' operators are overloaded to achieve independence of the implementation (definitions of member functions) from the interface (prototypes of member functions).

Overloading the new and the delete Operators

The 'new' and the 'delete' operators can be overloaded for specific classes. The behavior of these operators can be altered for operands of specific class types.

If these operators are overloaded for a specific class, then the functions that overload them are called when the class type is passed as a parameter to these operators. Otherwise, the global 'new' and 'delete' operators are called. For example, if the 'new' operator has been overloaded for a class X but not for a class Y, then the statement

```
X * XPtr = new X;
```

will call the function that overloads the 'new' operator of class X. But the statement

```
Y * Yptr = new Y;
```

will call the global 'new' operator. It is interesting to note that the user programs may not change if the functions to overload the 'new' and the 'delete' operators are inserted into a class or removed from it.

The syntax for overloading the 'new' operator (for allocating memory for a single object) is as follows:

```
class <class_name>
{
  public:
    static void * operator new(size_t); //function
                                        //prototype
    /*
      rest of the class
    */
};

void * <class_name> :: operator new ( size_t size)
                                  //function definition
{
  /*
    definition of the function
  */
}
```

Listing 8.39 Syntax for functions that overload the 'new' operator and allocate memory for a single object

The syntax for overloading the 'new' operator (for allocating memory for an array of objects) is as follows:

```
class <class_name>
{
  public:
    static void * operator new [](size_t);    //function
                                              //prototype
    /*
      rest of the class
    */
};
void * <class_name>::operator new [] ( size_t size)
                                    //function definition
{
  /*
    definition of the function
  */
}
```

Listing 8.40 Syntax for functions that overload the 'new' operator and allocate memory for an array of objects

The syntax for overloading the 'delete' operator (for deallocating memory for a single object) is as follows:

```
class <class_name>
{
  public:
    static void operator delete(void *,size_t);
                                    //function prototype
    /*
      rest of the class
    */
};

void <class_name>::operator delete (void * p, size_t size)
                                    //function definition
{
  /*
    definition of the function
  */
}
```

Listing 8.41 Syntax for functions that overload the 'delete' operator and deallocate memory for a single object

The syntax for overloading the 'delete' operator (for deallocating memory for an array of objects) is as follows:

```
class <class_name>
{
  public:
    static void operator delete [] (void *, size_t );
                                    //function prototype
    /*
      rest of the class
    */
};

void <class_name>::operator delete [](void *, size_t size)
                                    //function definition
{
  /*
    definition of the function
  */
}
```

Listing 8.42 Syntax for functions that overload the 'delete' operator and deallocate memory for an array of objects

The operator new function and the operator delete function must be static. However, their prototypes may or may not be prefixed with the `static` keyword. Either way, the compiler treats these functions as static (reasons for this are explained later in this chapter).

The return type of the operator new function must be of type `void *`. The value returned by this function is the address of the memory block it captures by calling the global new operator. The operator new function should take at least one formal argument of type `size_t`. As we will discover later, the operator new function can take more than one formal argument also. The `size_t` argument holds the amount of memory to be allocated in bytes. The code in Listing 8.43 illustrates this. It also illustrates how the global new operator is used from within the member new operator function to capture memory in the heap area and how a pointer to that memory is returned.

```
/*Beginning of A.h*/
class A
{
    int x;
  public:
    static void * operator new(const size_t);
    /*
      rest of the class A
    */
};
/*End of A.h*/
```

```
/*Beginning of A.cpp*/
#include"A.h"
void * A :: operator new(const size_t size)
{
  cout << sizeof(A) << endl;
  cout << size << endl;
  void * p = :: operator new(size);
  return p;
}
/*
  definitions of the rest of the functions of class A
*/
/*End of A.cpp*/

/*Beginning of Test.cpp*/
#include"A.h"
void main()
{
  A * APtr = new A;
}
/*End of Test.cpp*/
```

Output

4
4

Listing 8.43 Overloading the new operator

It is obvious that the class designer will overload the 'new' operator only if he/she is not satisfied with the default action of the 'new' operator for his/her class and would therefore like to fine-tune it. For this, he/she will insert the necessary code in the function to overload the 'new' operator. Apart from this code, statements to allocate the required amount of memory (by calling the global new operator) and then to return the address of the captured memory block are also inserted in the function to overload the 'new' operator. Otherwise, the requested memory will never get allocated.

The return type of the operator delete function must be `void` as it does not return anything. Its first formal argument should be of type `void *`. The address of the memory block being deleted is passed to it. The second formal argument of the operator delete function is of type `size_t`. The size of the memory block to be deleted is passed as a parameter to it. Listing 8.44 illustrates all this. It also illustrates how the global delete operator should be used to deallocate the memory being targeted.

```
/*Beginning of A.h*/
class A
{
    int x;
  public:
    static void operator delete(void * const,
                                const size_t );
    /*
      rest of the class A
    */
};
/*End of A.h*/

/*Beginning of A.cpp*/
#include"A.h"
void A :: operator delete(void * const p,
                          const size_t size )
{
  cout << p << endl;
  cout << sizeof(A) << endl;
  cout << size << endl;
  ::operator delete(size);
}
/*
  definitions of the rest of the functions of class A
*/
/*End of A.cpp*/

/*Beginning of Test.cpp*/
#include"A.h"
void main()
{
  A * APtr = new A;
  cout << APtr << endl;
  delete APtr;
}
/*End of Test.cpp*/
```

Output

0xCCCCCC
0xCCCCCC
4
4

Listing 8.44 Overloading the 'delete' operator

The reason for overloading the 'delete' operator is similar to the reason for overloading the 'new' operator. As in the case of the 'new' operator, the class designer will overload the 'delete' operator only if he/she is not satisfied with the default action of the 'delete' operator for his/her class and would therefore like to fine-tune it. For this, he/she will insert the necessary code in the function to overload the 'delete' operator. Apart from this code, a statement to deallocate the required amount of memory (by calling the global delete operator) is also inserted in the function to overload the 'delete' operator. Otherwise, the requested memory will never get deallocated.

The operator new function and the operator delete functions are static by default. This means that the compiler treats them as static functions whether the class designer uses the `static` keyword in their declarations or not. This is because the compiler places a call to the constructor immediately after the call to the 'new' operator and a call to the destructor immediately before the call to the 'delete' operator. The statement

```
A * APtr = new A;
```

is translated by the compiler as

```
A * APtr = A::operator new(sizeof(A)); //nameless object
                                       //created
A::A(APtr); //constructor called for nameless object
```

While the statement

```
delete APtr;
```

is translated by the compiler as

```
A::~A(APtr);//destructor called for nameless object
A::operator delete(APtr, sizeof(A));//nameless object
                                    //destroyed
```

In order to understand the implications of these translations, let us consider a class that has a pointer as one of its data members. The class designer would certainly like to initialize this pointer to some valid value (say, NULL) in the constructor of the class.

```
class String
{
    char * cStr;
  public:
    String()
    {
      cStr = NULL;
      /*
```

```
        rest of the function String::String()
      */
    }
    /*
      rest of the class String
    */
};
```

However, if access is allowed to the private data members in the new operator function, the class designer may accidentally allocate some memory dynamically in the heap and make 'cStr' point at it.

```
void * String::operator new(const size_t size)
{
  . . . .
  cStr = new char ...
  . . . .
}
```

Now when the constructor is called, a memory leak will occur because the value of 'cStr' will be straight away nullified without first deallocating the memory block at which it is pointing. We may suggest the following improvisation (Listing 8.45) in the code for the class constructor:

```
A::A()
{
  if(cStr != NULL)
    delete [] cStr;
  cStr = NULL;
}
```

Listing 8.45 The 'new' and 'delete' operators are static

However, there is a serious drawback in this code. It presupposes that if 'cStr' is not NULL, then it is definitely pointing at a dynamically allocated memory block that has been captured earlier by using the 'new' operator. This is true only if the objects are created by using the 'new' operator. However, if objects are created in the normal fashion as follows:

```
A A1;
```

then the mere fact that the code in the constructor of the class 'String' finds that the value of 'cStr' is not NULL does not mean that 'cStr' is definitely pointing at a valid block of

memory. In fact, in this case, 'cStr' is simply a rogue pointer. Using the 'delete' operator on such a rogue pointer will naturally lead to a run-time error. For such reasons, the C++ compiler prevents access to the non-static data members of the class by treating the operator new and operator delete functions as static. It is repeated that the compiler treats these functions as static whether we mention the `static` keyword in its declaration or not. We cannot force access to the non-static data members in the new and delete operator functions by avoiding the `static` keyword in their declarations.

It will not be out of context to mention once again that the constructor does not 'construct' the object, that is, it does not actually allocate memory for the object. It is merely a function that is called immediately after the memory for the object has actually been allocated. Its job is to ensure guaranteed initialization of all data members to proper values and to acquire any resources if necessary. Similarly, the destructor does not actually destroy the object in the sense that it does not actually deallocate the memory block occupied by the object. It is merely a function that is called immediately before the memory for the object is deallocated. Its job is to ensure a proper clean-up operation and to release all resources that were acquired during the lifetime of the object. The manner in which the compiler translates calls to the 'new' and 'delete' operators makes all this amply clear. Moreover, we may note how the global new and delete operators are called from the class member functions that overload them.

Values are passed to the constructor in the usual way even after the 'new' operator is overloaded.

```
A * APtr = new A(10,20);
```

is translated by the compiler as

```
A * APtr = A::operator new;
A::A(APtr,10,20);
```

Listing 8.46 Passing parameters to an overloaded 'new' operator.

We have already discussed, in brief, an overall reason for overloading the 'new' and 'delete' operators. The class designer overloads these operators if he/she considers their default action inappropriate or inefficient for his/her class. Now, we will learn about the specific cases where overloading these operators becomes beneficial.

First, let us see how the 'new' operator works. In order to deallocate the correct amount of memory, the 'delete' operator must know how much memory the 'new' operator has allocated. The compilers solve this problem by prefixing the memory block allocated by

the 'new' operator with the amount of memory allocated. Therefore, as a result of the statement

```
A * APtr = new A;
```

we get Part A and not Part B of Diagram 8.2 (suppose objects of class A occupy 10 bytes)

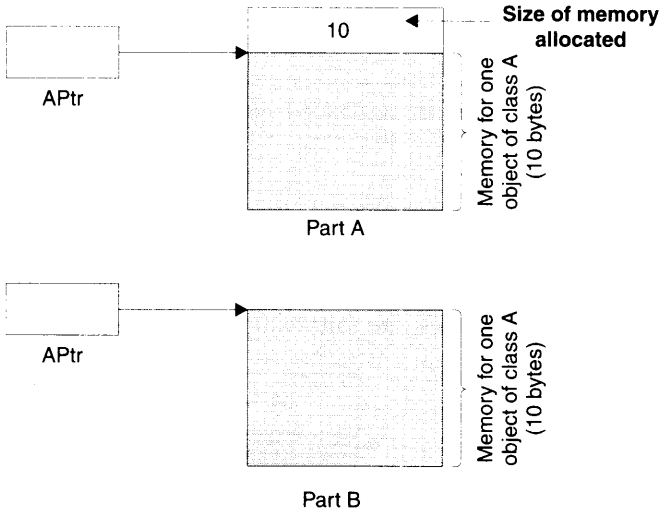

Diagram 8.2 Wastage of memory due to the default action of the 'new' operator

This means that every time the 'new' operator is called for the class A, a separate block to store the amount of memory allocated will also be allocated. This consumption of memory can be a real bottleneck in applications where memory is critical.

Before we come to a solution to this problem, we should know that even if the 'new' operator allocates an array of objects, only one memory slice will be prefixed to the allocated memory block in order to hold its size. Therefore, the amount of extra memory allocated remains the same whether one object is created in the heap or an array of objects is created in the heap. The class designer can take advantage of this fact.

The class designer can ensure that when the 'new' operator is called for the first time, a memory to hold a large number of objects gets allocated and the address of the memory block gets returned. The address returned will obviously be the address of the first object in the pool. Thereafter, every call to the 'new' operator will return the address of the next available block from the pool.

This solution is explained in Diagrams 8.3 and 8.4. Suppose A is a class whose objects occupy 10 bytes. The class designer reckons that the pool will hold five objects at a time. Therefore, the pool size will be equal to 50 bytes. Let us consider the following piece of code:

```
. . . .
. . . .
A * APtr01 = new A;       //line 1
A * APtr02 = new A;       //line 2
A * APtr03 = new A;       //line 3
A * APtr04 = new A;       //line 4
A * APtr05 = new A;       //line 5
A * APtr06 = new A;       //line 6
. . . .
. . . .
```

Listing 8.47 Calling an overloaded 'new' operator

A memory pool of 50 bytes (10 bytes for each of the five objects) will be created when line 1 executes (because the 'new' operator is being called for the first time). When lines 2 to 5 execute, addresses of adjacent blocks are returned sequentially from this pool. After the first line executes, the following scenario emerges:

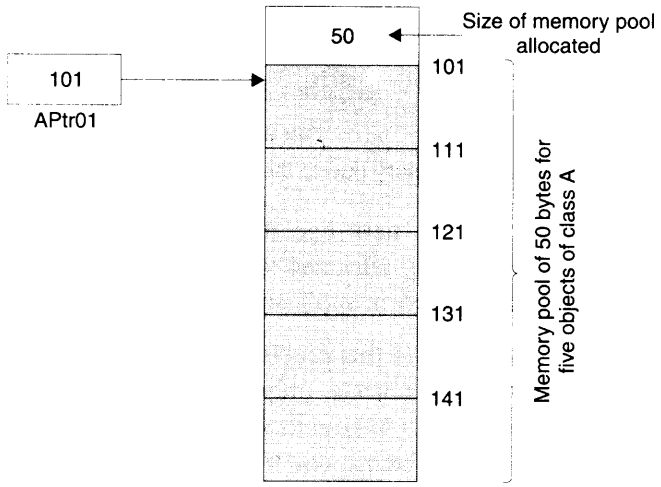

Diagram 8.3 Saving memory by overloading the 'new' operator and modifying its behavior

The address of the allocated memory block is '101' (say). Therefore, the value of 'APtr01' becomes '101'. The address block from '101' to '150' has been allocated as a result of the first line. As we can see, the size of the memory block (50) has been prefixed to the memory block itself. The second line will not cause another memory block to be allocated. Instead, the address of the next segment from the memory pool having address '111' will be returned. Therefore, the value of 'APtr02' will become '111'. The following scenario will emerge:

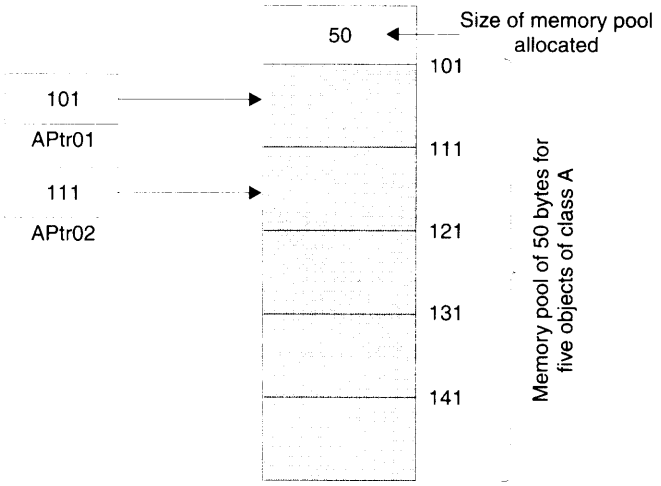

Diagram 8.4 Accessing the memory pool allocated by the overloaded 'new' operator

This process will continue until the sixth line is reached. After the fifth line finishes execution, the memory pool will get exhausted. At this point of time, another memory pool of the same size (50 bytes—10 bytes for five objects each) will get allocated and the process will repeat itself. All this is the effect of the code the class designer has written in the function that overloads the 'new' operator.

The actual code is given below:

```
/*Beginning of NewForMemorySave.h*/
class  ClassNew
{
    union
    {
      int x;
      ClassNew * next;
    }v;
    static int NO_OF_OBJECTS;
    static ClassNew * head;
  public:
    void setx(const int = 0);
    int getx() const;
    static void * operator new(const size_t);
};
/*End of NewForMemorySave.h*/

/*Beginning of NewForMemorySave.cpp*/
#include"NewForMemorySave.h"

int ClassNew::NO_OF_OBJECTS = 5;
```

```
ClassNew * ClassNew::head;

int ClassNew::getx() const
{
  return v.x;
}

void ClassNew::setx(const int p)
{
  v.x=p;
}
void * ClassNew::operator new(const size_t size)
{
  ClassNew * temp,*p;
  temp = head;
  if(!temp)
  {
    temp = (ClassNew *)::operator new(sizeof(class
           ClassNew)*NO_OF_OBJECTS);
    for(p=temp;p!=&temp[NO_OF_OBJECTS-1];p++)
      p->v.next=p+1;
    p->v.next=0;
  }
  head=temp->v.next;
  return temp;
}
/*End of NewForMemorySave.cpp*/

/*Beginning of NewForMemorySaveMain.cpp*/
#include<iostream.h>
#include"NewForMemorySave.h"
void main()
{
  ClassNew * ClassNewPtr01 = new ClassNew;
  ClassNewPtr01->setx(10);
  cout<<ClassNewPtr01->getx()<<endl;
  ClassNew * ClassNewPtr02 = new ClassNew;
  ClassNewPtr02->setx(20);
  cout<<ClassNewPtr02->getx()<<endl;
  ClassNew * ClassNewPtr03 = new ClassNew;
  ClassNewPtr03->setx(30);
  cout<<ClassNewPtr03->getx()<<endl;
  ClassNew * ClassNewPtr04 = new ClassNew;
  ClassNewPtr04->setx(40);
  cout<<ClassNewPtr04->getx()<<endl;
  ClassNew * ClassNewPtr05 = new ClassNew;
  ClassNewPtr05->setx(50);
  cout<<ClassNewPtr05->getx()<<endl;
  ClassNew * ClassNewPtr06 = new ClassNew;
```

```
  ClassNewPtr06->setx(60);
  cout<<ClassNewPtr06->getx()<<endl;
}
/*End of NewForMemorySaveMain.cpp*/
```

Output

```
10
20
30
40
50
60
```

Listing 8.48 Overloading the 'new' operator to improve efficiency

When this program runs, first memory for the static data members of class 'ClassNew' gets allocated. The static data member 'NO_OF_OBJECTS' stores how many blocks (objects) will coexist in each pool. This data member should be static because it contains information for the set of objects and is therefore not particular to any specific object. The value of this data member should be chosen with care. A value that is too large will waste memory and thereby prove counterproductive. Large portions of the pool may remain unutilized for long periods or for the entire lifetime of the program. A very small value will necessitate a frequent allocation of more pools, thereby slowing down the program. If it is felt that large number of objects will exist simultaneously at any given point of time, the value of this variable should be kept large, otherwise this value should be small. In our case, we have initialized 'NO_OF_OBJECTS' to '5'.

Next, memory for another static data member 'head' is allocated. This pointer points at the next available block from the pool. This data member should also be static because it will function for the entire set of objects and will, therefore, not be a part of any particular object. Every call to the 'new' operator returns the current value of this pointer and increments its value so that it then points at the next available block in the pool. This pointer has been initialized to NULL for reasons that will soon become apparent.

Now the main function begins execution. Memory for 'ClassNewPtr01' (four bytes since it is a pointer) is allocated. Currently, it has junk value. Next, the 'new' operator function for class 'ClassNew' is called. Two pointers, 'temp' (for holding the current value of 'head' so that 'head' can move to the next block) and 'p' (for traversing through the pool) are created. The pointer 'temp' is initialized to the current value of head. Now, the current value of the 'head' pointer is evaluated. During the first call to the 'new' operator, its value will be NULL. We will soon see that its value will be NULL under a slightly

different circumstance also. The test expression in the `if` construct succeeds. Therefore, the `if` block executes. Sufficient memory for holding 'NO_OF_OBJECTS' number of objects is allocated by calling the global 'new' operator. The address of this pool (in other words the address of the first object or block in this pool) is then stored in the 'temp' pointer. After this, the `for` loop makes the value of the 'next' pointer of each object (except the last object) in the newly created pool equal to the address of the next object. After the loop terminates, the last statement of the `if` block makes the 'next' pointer of the last object NULL. With this, the `if` block terminates. The second last statement of the function makes the 'head' pointer point at the second object of the pool. The value of 'temp', that is, the address of the first object of the pool is then returned by the operator new function. Thus, 'ClassNewPtr01' now points at the first object of the pool. 'ClassNewPtr01' pointer now operates on the first object of the pool by calling the member functions of the class 'ClassNew'.

Now let us look at the fourth statement of the main function in Listing 8.48. The operator new function of class 'ClassNew' will be called for a second time. Again, 'temp' pointer will be initialized to the current value of 'head' which is not NULL ('head' is pointing at the second object of the pool). The `if` block will therefore be skipped. Only the last two statements execute. The 'head' pointer is again incremented to point at the next object (in this case the third object) and the address of the object (in this case the second object) at which 'temp' is currently pointing is returned.

This process will continue till the operator new function of the class 'ClassNew' is called for the fifth time (`ClassNew * ClassNewPtr05 = new ClassNew`). Although this time also the test expression of the `if` block will fail, the value of the 'head' pointer will become NULL while the address of the fifth and last block in the pool will be returned. Now, when the operator new function is called for the sixth time, the test expression of the `if` block succeeds. A fresh pool is allocated and the process repeats itself.

In order to make the operator new function succeed, it is necessary to thread the memory by using the `for` loop. This necessitates the presence of the 'next' pointer in each object of the pool. Wastage of memory because of this 'next' pointer, which would otherwise defeat the very purpose for which the 'new' operator was overloaded, is elegantly prevented by the use of a `union`.

Now let see how we can overload the 'delete' operator for the class 'ClassNew'. The 'delete' operator will be overloaded in a manner that will not actually deallocate the memory. Instead, the block at which the pointer, on whom the delete operator is being applied points, will be put back in the free list. When the 'new' operator is called for the next time, the address of that block will be returned. Before looking at the actual code, let us see its effects.

Let us consider the case where the 'new' operator is called once only. This solitary call is followed by a call to the 'delete' operator.

```
#include"NewForMemorySave.h"
#include<iostream.h>
void main()
{
  ClassNew * ClassNewPtr01 = new ClassNew;
  ClassNewPtr01->setx(10);
  cout<<ClassNewPtr01->getx()<<endl;
  delete ClassNewPtr01;
}
```

Listing 8.49 Calling an overloaded 'delete' operator

After the execution of the first statement, the following scenario emerges:

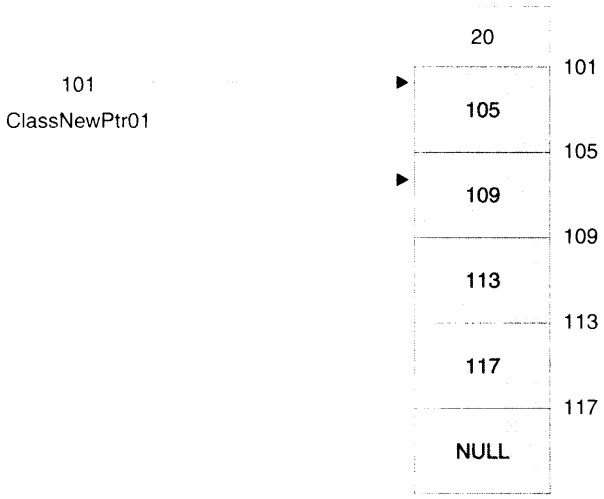

Diagram 8.5 Effect of the overloaded 'delete' operator

Memory for five objects gets allocated in a pool. The addresses of the blocks in the pool are displayed on the right, while the values of the 'next' pointers embedded in each object is shown within the rectangles that represent the objects. The amount of memory allocated (20 bytes for five objects @ four bytes per object) has been prefixed to the pool. After the second statement executes, the following scenario as shown in Diagram 8.6 emerges:

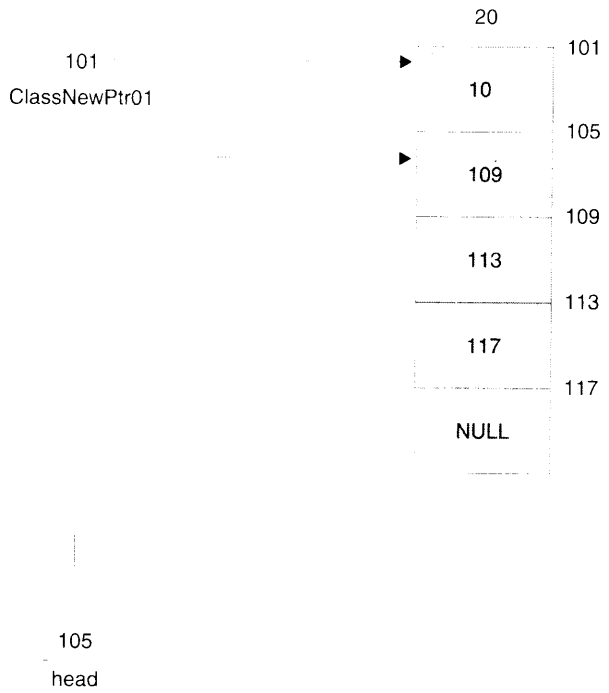

Diagram 8.6 Effect of the overloaded 'delete' operator

The value of the member 'x' of the first block is modified to '10'. The third statement does not alter the pool in any way. Now let us consider the fourth statement. The 'delete' operator is being applied on the pointer 'ClassNewPtr01'. It is pointing at the block with address '101'. The operator delete function will copy the current value of the 'head' pointer to the next pointer of this block. Thus, the 'next' pointer of the block will point at the head of the free list. Thereafter, the address of this block will be copied to the 'head' pointer. As a result, the scenario shown in Diagram 8.7 will emerge.

Now, if the 'new' operator is called, the address of the first block (101) will be returned by the operator new function.

Question: For the following two circumstances, try to explain this action of the operator delete function.

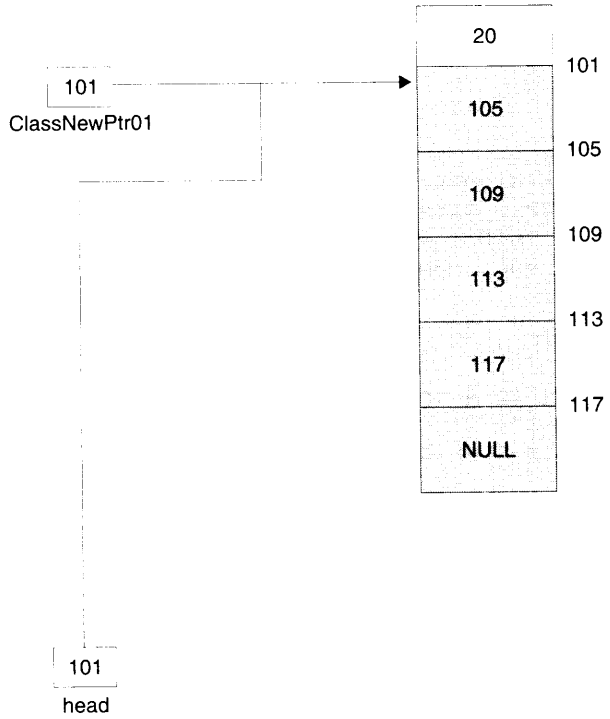

Diagram 8.7 Effect of the overloaded 'delete' operator

1. The pointer to be deleted and the 'head' pointer are not pointing to blocks that are next to each.
2. There are two pools (say six objects). The 'delete' operator is applied on the sixth pointer and then on the fifth.

The actual code is given below.

```
void ClassNew::operator delete(void * p, const size_t size)
{
  ClassNew * temp = (ClassNew)p;
  temp->next = head;
  head = temp;
}
```

Listing 8.50 Overloading the 'delete' operator

Let us end this discussion with a word of caution. Operator new and operator delete functions get inherited. This gives rise to bugs. We will soon see why this is so. However, first let us prove that this inheritance does occur.

```
/*Beginning of A.h*/
class A
{
    int x;
  public:
    void setx(const int = 0);
    int getx() const;
    static void * operator new(const size_t);
    static void operator delete(void * const, const
                                size_t);
};
/*End of A.h*/
/*Beginning of A.cpp*/
#include"A.h"
#include<iostream.h>

void A::setx(const int p)
{
  x=p;
}

int A::getx() const
{
  return x;
}

void * A::operator new(const size_t size)
{
  cout<<"operator new of class A called\n";
}

void A::operator delete(void * const p, const size_t size)
{
  cout<<"operator delete of class A called\n";
}

/*End of A.cpp*/

/*Beginning of B.h*/
#include"A.h"
class B : public A
{
    int y;
  public:
    void sety(const int = 0);
    int gety() const;
};
/*End of B.h*/
```

```
/*Beginning of B.cpp*/
#include"B.h"

void B::sety(const int q)
{
  y=q;
}

int B::gety() const
{
  return y;
}

/*End of B.cpp*/

/*Beginning of Main.cpp*/
#include"B.h"
void main()
{
  B * BPtr01 = new B;
  delete BPtr01;
}
/*End of Main.cpp*/
```

Output

operator new of class A called
operator delete of class A called

Listing 8.51 Operator new and delete functions get inherited

Thus, when the 'new' and 'delete' operators are called by passing the derived class type as parameter, it is seen that the 'new' and 'delete' operator functions of the base class are called. How can this be a problem?

The problem this inheritance causes is to due the fact that when the 'new' operator is called, the 'head' pointer points at the wrong place. Let us consider the overloaded 'new' operator in Listing 8.48. Suppose the 'new' operator is overloaded for class A in Listing 8.51 in the same way it is overloaded for class 'ClassNew' in Listing 8.48. After the 'new' operator executes for the first time, the head pointer points four bytes away from the first byte of the pool even if the derived class type is passed as a parameter to the 'new' operator. Thus, when the 'new' operator is called for the second time, the address of the fifth byte is returned and not the ninth byte. The address of the ninth object is desired because an object of the class B will occupy eight bytes—four for 'x' and four for 'y'. The problem that arises because of this can be clearly understood from the following code Listing 8.52.

```
/*Beginning of Main.cpp*/
#include"B.h"
void main()
{
  B * BPtr01 = new B;
  B * BPtr02 = new B;
  BPtr02->setx(20);
  cout<<BPtr02->getx()<<endl;
  BPtr01->sety(10)<<endl;
  cout<<BPtr02->getx()<<endl;
}
/*End of Main.cpp*/
```

Output

20
10

Listing 8.52 Undesirable effect of operator new and delete functions getting inherited

After the first two statements in Listing 8.52 execute, 'BPtr01' will point at the first byte of the memory pool and 'BPtr02' will point at the fifth. The third statement writes 20 into the block of bytes from the fifth byte to the eighth byte. However, the fifth statement writes '10' into the same memory block!

In order to neutralize this effect of inheritance, the size of memory block being targeted for allocation or deallocation should be compared with the size of the class for which the operator is being overloaded. If these two do not match, then the global 'new' or the global 'delete' operator should be called.

```
void * A::operator new(const size_t size)
{
  if(size != sizeof(class A)) //true if derived class type
                              //passed as parameter
    return ::operator new(size);
  //rest of the code
}

void A::operator delete(void * const p, const size_t size)
{
  if(size != sizeof(class A)) //true if derived class type
                              //passed as parameter
```

```
    {
      ::operator delete(p);
      return;
    }
    //rest of the code
}
```

Listing 8.53 Preventing the ill effects of the 'new' and 'delete' operators getting inherited

Overloading the Subscript Operator

The syntax for overloading the 'subscript' operator is as follows:

```
class <class_name>
{
  public:
    <return_type> operator[](<param_list>);  //prototype
};

//definition
<return_type> <class_name> :: operator[](<param_list>)
{
  //statements
}
```

Listing 8.54 Syntax for overloading the 'subscript' operator

The function that overloads the 'subscript' operator must be a non-static member of the class.

Let us overload the 'subscript' operator for the class 'String'. We will define the operator so that it returns the character stored in the position that is passed as a parameter to it.

```
/*Beginning of String.h*/
class String
{
  public:
    char& operator[](const int);
    /*
      rest of the class String
    */
};
/*End of String.h*/
```

```
/*Beginning of String.cpp*/
#include"String.h"
char& String :: operator[] (const int p)
{
  if(p<0 || p>len-1)
    throw "Invalid Subscript";
  return cStr[p];
}
/*
  definitions of the rest of the functions of class String
*/
/*End of String.cpp*/

/*Beginning of StringMain.cpp*/
#include"String.h"
#include<iostream.h>
void main()
{
  String s("abcd");
  cout<<s.getString()<<endl;
  cout<<s[0]<<endl;
  s[1]='x';
  cout<<s.getString()<<endl;
}
/*Beginning of StringMain.cpp*/
```

Output

abcd
a
abxd

Listing 8.55 Overloading the 'subscript' operator for the class 'String'

For the time being, we must ignore the `throw` statement (explained in Chapter 10) within the 'String :: operator[]()' function. The definition of the 'String :: operator[]()' function is quite simple. It finds out whether the subscript passed is within the acceptable limits or not. If not, it throws an exception. As we will learn in Chapter 10, throwing exceptions is a very effective and efficient way of error handling. If the subscript passed is within acceptable limits, the function returns the corresponding element by reference. Why is the element returned by reference? This is because the 'subscript' operator might be used on the left-hand side of the 'assignment' operator also. Under such circumstances, returning by reference causes the returned element to be assigned to the value passed on the right-hand side of the 'assignment' operator.

```
s[1]='x'; //assign 'x' to the second character in the
          //string held by s.
```

However, the definition of the 'String :: operator[]()' function has a flaw. Suppose there is a constant object.

```
const String s("abcd");
```

Now, if we call the 'subscript' operator with respect to the constant object, the compiler correctly throws a compile-time error.

```
cout << s[1] << endl;     //ERROR!
```

This is because the 'String :: operator[]()' function is not a constant function and therefore cannot be called with respect to the constant object. Let us therefore introduce another constant function that overloads the 'subscript' operator in the same way as the non-constant function.

```
/*Beginning of String.h*/
class String
{
  public:
    char& operator[](const int);    //for non-constant
                                    //objects
    char& operator[](const int) const;  //for constant
                                        //objects
    /*
      rest of the class String
    */
};
/*End of String.h*/

/*Beginning of String.cpp*/
#include"String.h"
char& String :: operator[] (const int p) //for non-constant
                                         //objects
{
  if(p<0 || p>len-1)
    throw "Invalid Subscript";
  return cStr[p];
}
char& String :: operator[] (const int p) const//for
                                       //constant objects
```

```
{
  if(p<0 || p>len-1)
    throw "Invalid Subscript";
  return cStr[p];
}

/*
  definitions of the rest of the functions of class String
*/
/*End of String.cpp*/
```

Listing 8.56 Overloading the 'subscript' operators for constant objects

Now, there are separate functions to overload the 'subscript' operator for constant and non-constant objects. However, the constant function given in Listing 8.56 is still imperfect. Let us consider the following piece of code.

```
const String s("abcd");
s[1]='x'; //unacceptable, but the compiler doesn't
          //complain!
```

The second statement in the code calls the constant function with respect to the constant object. The function returns the selected element by reference and the value of the selected element gets set to 'x'. Although the compiler will compile, our perception of a constant tells us that the second statement above should not compile. In order to ensure this, we must make the constant 'String :: operator[]()' function return the value as a constant reference and not as a non-constant reference.

```
/*Beginning of String.h*/
class String
{
  public:
    char& operator[](const int);    //for non-constant
                                    //objects
    const char& operator[](const int) const; //for constant
                                             //objects
    /*
      rest of the class String
    */
};
/*End of String.h*/

/*Beginning of String.cpp*/
#include"String.h"
char& String :: operator[] (const int p) //for non-constant
                                         //objects
```

```
{
  if(p<0 || p>len-1)
    throw "Invalid Subscript";
  return cStr[p];
}
const char& String :: operator[] (const int p) const
                                          //for constant
                                          //objects
{
  if(p<0 || p>len-1)
    throw "Invalid Subscript";
  return cStr[p];
}

/*
  definitions of the rest of the functions of class String
*/
/*End of String.cpp*/
```

Listing 8.57 Returning a constant value for constant objects

As desired, statements such as

```
s[1]='x';
```

will no longer compile if the calling object 's' is a constant.

We conclude this section on overloading the 'subscript' operator with one last piece of information. The formal argument of the function that overloads the 'subscript' operator can be of any type. In the example given in Listing 8.57, the formal argument was of type `const int`. However, it can be of any type, such as `char`, `float`, `double`.

```
/*Beginning of String.h*/
class String
{
  public:
    int operator[](const char);     //for non-constant
                                    //objects
    int operator[](const char) const;   //for constant
                                        //objects
    /*
      rest of the class String
    */
};
/*End of String.h*/
```

Listing 8.58 Formal argument of the function that overloads the 'subscript' operator can be of any type

Overloading the Pointer-to-member (->) Operator (Smart Pointer)

Overloading the pointer-to-member (->) operator is slightly complicated. First, let us understand that it is a postfix unary operator. If it is overloaded as follows:

```
class A
{
  public:
    B * operator->();
    /*
      rest of the class A
    */
};
```

then the second statement that follows

```
A p;
p->abc();
```

translates to

```
(p.operator->())->abc();
```

Obviously, 'abc()' must be a member of class B.

The pointer-to-member (->) operator can be overloaded to create 'smart pointers'.

'Smart pointers', unlike the ordinary 'unsmart' pointers, can be designed to inevitably point at valid objects. In contrast, the ordinary 'unsmart' pointers have to be explicitly initialized by the client. Consequently, they have to be tested for validity before every use. 'Smart pointers' are class objects. Let us create a class whose objects will behave like pointers to the class 'String'.

```
/*Beginning of StrPtr.h*/
#include"String.h"
class StrPtr
{
    String * p;
  public:
    StrPtr(String&);      //the one and only one constructor
};
/*End of StrPtr.h*/

/*Beginning of StrPtr.cpp*/
#include"StrPtr.h"
StrPtr::StrPtr(String& ss)
```

```
{
  p=&ss;
}
/*End of StrPtr.cpp*/
```

Listing 8.59 A class of smart pointers

We must notice how a zero-argument constructor has been deliberately left out from the class definition in Listing 8.59. This forces clients to invariably pass an object of the class 'String' as a parameter whenever they create objects of the class 'StrPtr'. Therefore, the embedded pointer of the class 'StrPtr' always points at an object of the class 'String'. The client can create an object of the class in Listing 8.59 as follows:

```
String s1("abc");
StrPtr p(s1);
```

To mimic a pointer completely, objects of the class 'StrPtr' should be capable of being used as follows:

```
p->setString("def");
```

For this, the pointer-to-member operator (->) needs to be overloaded for the class 'StrPtr'. This can be done as follows:

```
/*Beginning of StrPtr.h*/
#include"String.h"
class StrPtr
{
    String * p;
  public:
    StrPtr(String&);      //the one and only one constructor
    String * operator->();    //the overloaded operator
};
/*End of StrPtr.h*/

/*Beginning of StrPtr.cpp*/
#include"StrPtr.h"
StrPtr::StrPtr(String& ss)
{
  p=&ss;
}
String * StrPtr::operator->()
```

```
{
  return p;
}
/*End of StrPtr.cpp*/
```

Listing 8.60 Overloading the pointer-to-member operator for smart pointers

Contrast objects of the class 'StrPtr' with ordinary pointers that point at objects of the class 'String'. In case of ordinary pointers, there is no guarantee that the pointer being used is pointing at a valid block of memory.

```
void f1(String * p)
{
  p->setString("abc");//No way to check the validity of p
}
```

On the other hand, an attempt to similarly initialize an object of the class 'StrPtr' results in a compile-time error.

```
StrPtr p; //ERROR: no zero-argument constructor
```

Therefore, in case of smart pointers, there is a guarantee that the pointer being used is pointing at a valid block of memory.

```
void f1(StrPtr p)
{
  p->setString("abc");    //p is definitely valid
}
```

8.3 Type Conversion

In this section, we shall be dealing with techniques for converting variables from one type to another. Conversion of one type to another is achieved by the use of constructors and type-conversion functions.

Basic Type to Class Type

Conversion for basic type to class type is achieved by introducing a suitable constructor in the class. Suppose it is desired that the following statement should make 'd1.iFeet' equal to '1' and 'd1.fInches' equal to '9'.

```
Distance d1 = 1.75; //OR Distance d1(1.75);
```

A value ('1.75') which is of a basic type (`float`) needs to be converted into an object of the class 'Distance'. A suitable constructor in the class 'Distance' can carry out this conversion.

```
/*Beginning of Distance.h*/
class Distance
{
    int iFeet;
    float fInches;
  public:
    Distance(const float);
    /*
      rest of the class Distance
    */
};
/*End of Distance.h*/

/*Beginning of Distance.cpp*/
#include"Distance.h"
Distance::Distance(const float p)
{
  iFeet=(int)p;
  fInches=(p-iFeet)*12;
}
/*
  definitions of the rest of the functions of class
  Distance
*/
/*End of Distance.cpp*/
```

Listing 8.61 Using constructors for converting a value of basic type to class type

An ambiguity arises when two classes convert from the same type. Let us consider the following two classes:

```
class A
{
  public:
    A(int);
};

class B
{
  public:
    B(int);
```

```
};

void f(A);
void f(B);//function f() is overloaded

void g()
{
  f(1);   //ERROR: ambiguous call - f(X(1)) or f(Y(1))?
}
```

Listing 8.62 Ambiguity due to conversion from the same type

The ambiguity in Listing 8.62 can be resolved by an explicit type conversion:

```
f(X(1));  //OK
f(Y(1));  //OK
```

Class Type to Basic Type

Type-conversion operators achieve the conversion of class type to basic type. The syntax for the type-conversion functions is:

```
class <class_name>
{
  public:
    operator <type_name> ();  //prototype
    /*
      rest of the class
    */
};

<class_name> :: operator <type_name> () //definition
{
  /*
    definition of the function
  */
}
```

Listing 8.63 Syntax for converting values from class type to basic type

We must notice that the return type is not mentioned. Type-conversion operators resemble constructors in this respect. Let us introduce a function in the class 'Distance' for converting its objects into `float` type variables. In particular, we would like the value of the variable 'x' in the following piece of code to become '1.75'.

```
Distance d1(1,9);
float x=d1;
```

The code to achieve this transformation is as follows:

```
/*Beginning of Distance.h*/
class Distance
{
    int iFeet;
    float fInches;
  public:
    operator float();
    /*
      rest of the class Distance
    */
};
/*End of Distance.h*/
/*Beginning of Distance.cpp*/
#include"Distance.h"
Distance::operator float()
{
  return (iFeet+(fInches/12));
}
/*
  definitions of the rest of the functions of class
  Distance
*/
/*End of Distance.cpp*/
```

Listing 8.64 Converting from class type to basic type

Class Type to Class Type

Conversion of one class type value to another can be achieved by both a constructor and a type-conversion operator. Which of these two techniques will be used depends upon the class that is being provided the capability to convert the value.

If it is desired that the object on the left side of the assignment should have the ability, then a suitable constructor should be introduced in that object's class. If it is desired that

the object on the right side of the assignment should have the ability, then a suitable conversion operator should be introduced in that object's class.

```
class A {};

class B {};

void f()
{
  A A1;
  B B1;
  A1=B1;  //either class A should have a constructor or
          //class B should have a type conversion operator
}
```

Listing 8.65 Assigning an object of one class to another

The constructor can be introduced in class A as follows:

```
class A
{
  public:
    A(const B&);     //prototype
};

A::A(const B& b)     //definition
{
  /*
    definition of the function
  */
}
```

Listing 8.66 Using a constructor for converting from one class type to another

The type conversion operator can be introduced in class B as follows:

```
class B
{
  public:
    operator A();    //prototype
};
```

```
B::operator A()      //definition
{
  /*
    definition of the function
  */
}
```

Listing 8.67 Using a type conversion operator for converting from one class type to another

Care should be taken to ensure that only one of these two techniques is used on the pair of classes. If both are used together, the compiler throws an ambiguity error when the objects of the two classes are equated. This is because both the techniques can carry out the conversion and the compiler is not in a position to choose between the two.

8.4 New Style Casts and the typeid Operator

C++ provides a new set of operators for typecasting. These operators can be used instead of the highly error-prone method of typecasting provided by the C language. An example of the traditional method of typecasting was mentioned in the section titled 'Explicit address manipulation' in Chapter 2.

The new style casts are safe to use and can be easily located in source codes by using the search facility of the editor in which the source code has been opened. The latter benefit is especially useful in large source codes.

Ideally, a program should not need casts at all. However, there are various programming patterns where they are necessary. In order to meet this need, new style casts should be used instead of the old traditional style.

There are four new style cast operators.

- `dynamic_cast`
- `static_cast`
- `reinterpret_cast`
- `const_cast`

Each of the these operators converts the object, which is passed to it as an operand, in a pre-defined way and returns the converted object. The general syntax of these operators is:

```
operator <type>(value whose type is to be converted)
```

The `typeid` operator is similar to the `dynamic_cast` operator.

The *dynamic_cast* Operator

Run time type information (RTTI) enables us to find the type of a value and to compare the types of two values. C++ provides `dynamic_cast` operator and the `typeid` operator for implementing RTTI.

The `dynamic_cast` operator is used to determine whether a particular base class pointer points at an object of the base class or an object of one of the derived classes at run time. It is also used to determine whether a base class reference refers to an object of the base class or an object of one of the derived classes at run time.

We know from Chapter 5 that a base class pointer can point at an object of the derived class while a derived class pointer cannot point at an object of the base class.

Let A be a base class and B be its derived class.

```
A A1, * APtr;
B B1, * BPtr;
APtr=&B1;   //line 1: OK: Can convert from B* to A*
BPtr=&A1;   //line 2: ERROR: Cannot convert from A* to B*
BPtr=APtr; //line 3: ERROR: Cannot convert from A* to B*
```

However, in the first line of this piece of code, 'APtr' (a base class pointer) points at 'B1' (an object of the derived class). In this particular case, there should be no harm in assigning the value of the base class pointer to the derived class pointer (see the third line). After all, the base class pointer contains the address of a derived class object and a derived class pointer can certainly point at an object of the derived class.

However, a statement that assigns the value of a base class pointer to a derived class pointer (line 3) will not compile. The compiler has no way of knowing the type of the object whose address would get assigned to the base class pointer at run time.

As we already know, a pointer usually appears as function argument. Usually, it is not a local variable. The library programmer puts the prototypes of his/her functions, including the ones that have pointers as formal arguments, in header files and their compiled definitions in libraries. These functions are called from functions that are defined in application codes or in other library codes. In case the particular library function being called has a pointer as a formal argument, the application source code passes a suitable address to it. This address can be the address of a base class object or a derived class object. However, within the definition of the library function, there is no way of determining the exact type of the object whose address will be passed to it. Therefore, a line within the library function such as the third line in the foregoing code snippet will not compile.

However, the library programmer may need to assign the value of a base class pointer to a derived class pointer if the base class pointer points at an object of the same derived

class. The `dynamic_cast` operator enables us to know the type of the object whose address gets assigned to a base class pointer during run time.

Please refer to the general syntax of the new style cast operators given at the beginning of this section. If 'type' is a derived class pointer type and the value to be converted is the address of an object of the same derived class, then the `dynamic_cast` operator returns a pointer to the object. Else, it returns NULL. Remember that for the `dynamic_cast` operator to operate, the base class should be polymorphic in nature, that is, it should have at least one virtual function. An illustrative program follows:

```
/*Beginning of dynamicCast01.cpp*/
#include<iostream.h>
class A
{
  public:
    virtual void f1()
    {
      cout<<"A::f1() called\n";
    }
};

class B : public A
{
  public:
    void f2()
    {
      cout<<"B::f2() called\n";
    }
};

class C : public A
{
  public:
    void f3()
    {
      cout<<"C::f3() called\n";
    }
};

void main()
{
  A * Aptr;
  B B1, * BPtr;
  C C1;
```

```
    APtr=&B1;          //APtr points at an object of class B.
    BPtr=dynamic_cast<B*>(APtr);     //APtr is actually of
                                     //type B* and type is
                                     //also B*. Hence, cast
                                     //returns address of B1.
    if(BPtr!=NULL)     //BPtr is not NULL. It contains the
                       //address of B1.
      BPtr->f2();
    else

      cout<<"Invalid cast\n";

    APtr=&C1;          //APtr points at an object of class C.
    BPtr=dynamic_cast<B*>(APtr);     //APtr is actually of
                                     //type C* and type is
                                     //B*. Hence, cast
                                     //returns NULL.
    if(BPtr!=NULL)     //BPtr is NULL.
      BPtr->f2();
    else
      cout<<"Invalid cast\n";
}
/*End of dynamicCast01.cpp*/
```

Output
B::f(2) called
Invalid cast

Listing 8.68 Using the `dynamic_cast` operator with pointers

The process implemented in Listing 8.68 for safely casting a pointer of base class type to a pointer of derived class type is known as safe downcasting. This process enables us to access those features of the derived class that are not present in the base class.

If the `dynamic_cast` operator is used with references, it throws an exception of type 'Bad_cast' where it would have otherwise returned NULL, had pointers been used. Understanding this requires preliminary knowledge of exception handling. Therefore, the following listing can be read after reading the Chapter 10 on exception handling.

```
/*Beginning of dynamicCast02.cpp*/
#include<iostream.h>
#include<typeinfo.h>
```

```
class A
{
  public:
    virtual void f1()
    {
      cout<<"A::f1() called\n";
    }
};

class B : public A
{
  public:
    void f2()
    {
      cout<<"B::f2() called\n";
    }
};

class C : public A
{
  public:
    void f3()
    {
      cout<<"C::f3() called\n";
    }
};

void main()
{
  B BObj;
  C CObj;

  A & ARef1=BObj;    //ARef1 is a reference to an object of
                     //class B
  try
  {
    B & BRef1=dynamic_cast<B &>(ARef1); //ARef1 is actually
                                        //of type B& and
                                        //type is also B&.
                                        //Hence, cast
                                        //returns reference
                                        //to BObj.
    BRef1.f2();
  }
  catch(Bad_cast)
  {
    cout<<"Invalid cast\n";
  }
```

```
    A & ARef2=CObj;    //ARef1 is a reference to an object of
                       //class C
    try
    {
      B & BRef2=dynamic_cast<B &>(ARef2); //ARef2 is actually
                                          //of type C& and
                                          //type is B&.
                                          //Hence, cast
                                          //throws an
                                          //exception of type
                                          //Bad_cast.
      BRef2.f2();
    }
    catch(Bad_cast)
    {
      cout<<"Invalid cast\n";
    }
}
/*End of dynamicCast02.cpp*/
```

Output

B::f2() called
Invalid cast

Listing 8.69 Using the `dynamic_cast` operator with references

The *static_cast* Operator

The only difference between the `static_cast` operator and the `dynamic_cast` operator is that while the `dynamic_cast` operator carries out a run-time check to ensure a valid conversion (it returns NULL or throws an exception of type 'Bad_cast'), the `static_cast` operator caries out no such check.

```
/*Beginning of staticCast.cpp*/
#include<iostream.h>
class A
{
};

class B : public A
{
    int x;
```

```
  public:
    void setx(const int p)
    {
      cout<<"B::setx() called\n";
      x=p;
    }
    int getx()
    {
      return x;
    }
};

class C : public A
{
    int y;
  public:
    void sety(const int q)
    {
      cout<<"C::sety() called\n";
      y=q;
    }
    int gety()
    {
      return y;
    }
};

void main()
{
  A * APtr;
  B B1, *BPtr;
  C C1, *CPtr;

  APtr=&B1;
  BPtr=static_cast<B*>(APtr); //valid conversion; BPtr
                              //points at B1.
  BPtr->setx(1);
  cout<<B1.getx()<<endl;

  APtr=&C1;
  BPtr=static_cast<B*>(APtr);       //invalid conversion;
                                    //BPtr points at C1; but
                                    //static_cast operator
                                    //allows it.
  BPtr->setx(2);
  cout<<C1.gety()<<endl;
}
/*End of staticCast.cpp*/
```

Output

```
B::setx() called
1
B::setx() called
2
```

Listing 8.70 The `static_cast` operator

The first conversion by the `static_cast` operator in Listing 8.70 is correct. 'BPtr' (of type 'B*') points at 'B1' (of type B).

However, the second conversion by the `static_cast` operator is incorrect. 'BPtr' (of type 'B*') points at 'C1' (of type C). Since 'BPtr' is of type 'B*', the member functions of class B alone can be called with respect to it.

It is interesting to note what happens when 'BPtr' points at 'C1' and the 'B::setx()' function is called for it. The statement

```
x=p;
```

in 'B::setx()' function simply stores the value '2' in the first four bytes of the object at which 'BPtr' points. This is because 'B::x' is an integer type value and is the only data member of class B. However, these four bytes are occupied by 'C1.y'! Therefore, the output of the last statement in Listing 8.70 is '2'. The error-prone nature of the `static_cast` operator is quite evident from this.

However, this does not mean that the old style cast ('B*') is as good as the `static_cast` operator. The `static_cast` operator is still a better choice because it can be easily located in the source codes by searching for the string `static_cast`. Bugs suspected due to an invalid type conversion can thus be easily found out.

The *reinterpret_cast* Operator

Just like the old style cast, the `reinterpret_cast` operator allows us to cast one type to another.

Suppose 'cPtr' is a character pointer and 'vPtr' is a void pointer. If the value of 'vPtr' is to be assigned to 'cPtr', it needs to be typecast first.

```
cPtr=(char *)vPtr;
```

However, the preceding statement can be rewritten as

```
cPtr=reinterpret_cast<char *>(vPtr);
```

The compiler generates errors or warnings if casts are absent from conversion statements where a value of one type is being converted to an incompatible type. These errors and warnings can be switched off by inserting cast operators. Inserting a cast operator is a way of expressing our awareness and acceptance of the potential consequences to the compiler and the reader.

As in the case of `static_cast`, the `reinterpret_cast` operator seems to be an unnecessary substitute of the old style cast. Again, as in the case of `static_cast`, the visibility of the new style cast is considerably greater than the old style cast, which makes tracking down a rogue old style cast much easier.

The *const_cast* Operator

The `const_cast` operator serves the same purpose as the `mutable` keyword that has been explained in Chapter 2. The `const_cast` operator is used to cast away the constness of a pointer.

We may recall the following listing on mutable data members from Chapter 2 (Listing 2.21).

```
/*Beginning of mutable.h*/
class A
{
    int x;      //non-mutable data member
    mutable int y;  //mutable data member

  public:

    void abc() const      //a constant member function
    {
      x++;      //ERROR: cannot modify a non-mutable data
                //member in a constant member function
      y++;      //OK: can modify a mutable data member in a
                //constant member function
    }

    void def()            //a non-constant member function
    {
      x++;      //OK: can modify a non-mutable data member
                //in a non-constant member function
```

```
        y++;      //OK: can modify a mutable data member in a
                  //non-constant member function
    }
};
/*End of mutable.h*/
```

Listing 2.21 can be rewritten by using the `const_cast` operator instead of declaring the desired data member as mutable as follows:

```
/*Beginning of const_cast.h*/
class A
{
    int x;      //non-mutable data member
    int y;      //non-mutable data member

  public:
    void abc() const      //a constant member function
    {
      x++;      //ERROR: cannot modify a non-constant data
                //member in a constant member function
      const_cast<A*>(this)->y++;
                //OK: can modify a non-mutable data member
                //in a constant member function by casting
                //away the constness of the this pointer
    }

    void def()            //a non-constant member function
    {
      x++;      //OK: can modify a non-mutable data member
                //in a non-constant member function
      y++;      //OK: can modify a mutable data member in a
                //non-constant member function
    }
};
/*End of const_cast.h*/
```

Listing 8.71 The `const_cast` operator

The compiler treats the 'this' pointer as a constant pointer inside non-constant functions. However, it treats the 'this' pointer as a constant pointer to a constant inside constant functions. In the 'A::abc()' function in Listing 8.71, the constness of the 'this' pointer is cast away. This enables us to modify a non-mutable data member in a constant function.

We may note that by passing 'A*' to the `const_cast` operator in the 'A::abc()' function in Listing 8.71, the 'this' pointer was made an ordinary pointer that is neither a constant

nor supposed to point at a constant object. We could have very well passed 'A * const' instead and still ensured a successful compilation of the statement. This is because passing 'A * const' to the `const_cast` operator would have rendered the 'this' pointer a constant pointer that points at a non-const object.

The motive for using the `const_cast` operator is the same as the motive for using the `mutable` keyword.

As in the case of the other new style cast operators, using the `const_cast` operator indicates the programmer's awareness and acceptance of the possible negative consequences of its use.

The *typeid* Operator

Apart from the `dynamic_cast` operator, C++ provides the `typeid` operator for implementing RTTI (`typeid` is a keyword in C++). The `typeid` operator takes a value as its only parameter. It returns the type of the passed value as a reference to an object of class 'type_info'. The class 'type_info' is defined in the header file 'typeinfo.h'.

Two objects of the class 'type_info' can be compared by using the 'equality' operator. The name of the type of value passed to the `typeid` operator can also be determined by using the 'type_info::name()' function.

Values of fundamental data types, pointers to values of fundamental data types, and references to values of fundamental data types can be passed to the `typeid` operator.

```
/*Beginning of typeid.cpp*/
#include<typeinfo.h>
void main()
{
  char c;
  int i;
  float f;
  double d;

  cout<<typeid(c).name()<<endl;
  cout<<typeid(i).name()<<endl;
  cout<<typeid(f).name()<<endl;
  cout<<typeid(d).name()<<endl;

  if(typeid(i)==typeid(1.1))  //comparing int with float
    cout<<"i is of the same type as 1.1";
  else
```

```
    cout<<"i is not of the same type as 1.1";
}
/*End of typeid.cpp*/
```

Output

char
int
float
double
i is not of the same type as 1.1

Listing 8.72 The `typeid` operator

Class objects, pointers to class objects, or references to class objects can also be passed to the `typeid` operator. However, for the `typeid` operator to work correctly, the class whose object, pointer, or reference is passed to it should be polymorphic in nature. Otherwise, either of the following will happen depending upon the compiler and its settings:

(a) The compiler would issue a compile-time warning against the statement in which the `typeid` operator has been called. The OS would throw a run-time error.

(b) If a dereferenced base class pointer that points at a derived class object is passed as a parameter to the `typeid` operator, the `typeid` operator would not be able to determine the type of the object pointed at by the pointer. It would instead return the base class type as the type of the object pointed at by the base class pointer, which of course is undesirable.

```
class A {};       //no virtual function
class B : public A {};
B B1;
A * APtr=&B1;
cout<<typeid(*APtr).name()<<endl;     //prints: class A
```

Had the base class A in the preceding code contained at least one virtual function, the last 'cout' statement would have printed 'class B' as desired.

Given that class A does have a virtual function, what would the following tests evaluate to?

```
typeid(APtr) == typeid(A*)     //comparing pointers
typeid(*APtr) == typeid(A)     //comparing objects pointed
                               //at
```

The first of these test expressions would return true while the second one would return false. In the first case, the types of the pointers, and not the types of the objects being pointed at, are being compared. Since 'APtr' is of type 'A*', the first statement returns true. In the second case, the types of the objects being pointed at, and not the types of the pointers, are being compared. Since '*APtr' is of type B, the second statement returns false.

Summary

In C++, the library programmer can provide existing operators with additional capabilities to operate upon objects of his/her class. This is known as operator overloading.

Operators can be overloaded by functions having their names composed of the keyword `operator` and the symbol of the operator being overloaded. These functions may be member functions or friend functions. Friend functions are used when the objects of the class for which the operator is being overloaded invariably appear on the right-hand side of the operator.

Operators are overloaded to

- Neutralize the effect of the functions that are generated by default (the 'assignment' operator)
- To make the operation of the operators more efficient (the 'new' and 'delete' operators)
- To provide capabilities to the class so that its objects can be used in predefined templates.

The rules for operator overloading are as follows:

- New operators cannot be created.
- Meaning of existing operators cannot be changed.
- The following operators cannot be overloaded:

  :: (scope resolution)
  . (member selection)
  .* (member selection through pointer to member)
  ?: (conditional operator)
  sizeof (finding the size of values and types)
  typeid (finding the type of object pointed at)

- The following operators can be overloaded using member functions alone:

  = (Assignment operator)
  () (Function operator)
  [] (Subscripting operator)
  -> (Pointer-to-member access operator)

- Number of arguments that an existing operator takes cannot be changed.

The following type conversions can be carried out:

- basic type to class type (by using a constructor),
- class type to basic type (by using a type conversion operator), and
- class type to class type (by using either a constructor or a type conversion operator)

The C++ language provides a new set of operators for typecasting. These operators can be used instead of the highly error-prone method of typecasting provided by the C language.

The new style casts are safe to use and can be easily located in source codes by using the search facility of the editor in which the source code has been opened. The latter benefit is especially useful in large source codes.

There are four new style cast operators.

- `dynamic_cast`
- `static_cast`
- `reinterpret_cast`
- `const_cast`

Each of the new style cast operators converts the object that is passed to it as an operand in its own way and returns the converted object. The general syntax of these operators is:

```
operator <type>(value whose type is to be converted)
```

The `dynamic_cast` operator is used to determine whether a particular base class pointer points at an object of the base class or an object of one of the derived classes at run time. It is also used to determine whether a base class reference refers to an object of the base class or an object of one of the derived classes at run time.

In the general syntax, if 'type' is a derived class pointer type and the value to be converted is the address of an object of the same derived class, then the `dynamic_cast` operator returns a pointer to the object. Else, it returns NULL. For the `dynamic_cast` operator to operate, the base class should be polymorphic in nature, that is, it should have at least one virtual function.

The `dynamic_cast` operator enables us to access those features of the derived class that are not present in the base class.

If the `dynamic_cast` operator is used with references, it throws an exception of type 'Bad_cast' where it would have otherwise returned NULL had pointers been used.

While the `dynamic_cast` operator carries out a run-time check to ensure a valid conversion, the `static_cast` operator caries out no such check.

The `reinterpret_cast` operator allows us to cast one type to another.

New style casts are definitely a better choice than the old C-style casts. Visibility of the new style cast is considerably greater than the old style cast, which makes tracking down a rogue old style cast much easier.

The `const_cast` operator serves the same purpose as the `mutable` keyword. The `const_cast` operator is used to cast away the constness of a pointer.

Apart from the `dynamic_cast` operator, C++ provides the `typeid` operator for implementing RTTI (`typeid` is a keyword in C++). The `typeid` operator takes a value as its only parameter. It returns the type of the passed value as a reference to an object of class 'type_info'. The class 'type_info' is defined in the header file 'typeinfo.h'.

Two objects of the class 'type_info' can be compared by using the 'equality' operator. The name of the type of value passed to the `typeid` operator can also be determined by using the 'type_info::name()' function.

Class objects, pointers to class objects, or references to class objects can be passed to the `typeid` operator. However, for the `typeid` operator to work correctly, the class whose object, pointer, or reference is passed to it should be polymorphic in nature.

Key Terms

operator overloading

syntax for operator overloading

using friend functions for operator overloading

need for operator overloading

type conversions

- basic type of class type
- class type of basic type
- class type of class type

dynamic_cast operator

static_cast operator

reinterpret_cast operator

const_cast operator

typeid operator

Exercises

1. What is operator overloading?
2. How are operators overloaded?
3. How does the compiler interpret the operator-overloading functions?
4. Why are operators overloaded?
5. Under what circumstances does overloading using friend functions become necessary?
6. What is the difference between the functions that overload the increment operator in prefix and in postfix formats?
7. Why does the function to overload the 'assignment' operator receive and return by reference?
8. Explain why the function to overload the 'assignment' operator for the class 'String' returns `*this` and not the passed parameter.
9. Why is the 'assignment' operator function not inherited? Explain. Why does the compiler generate the 'assignment' operator for a class, for which the class designer has not defined one, and even if its base class already has the 'assignment' operator function implicitly or explicitly defined?
10. Why are objects of the classes 'istream' and 'ostream' passed and returned by reference in the functions to overload the 'insertion' and 'extraction' operators?
11. How is data abstraction achieved by overloading the 'insertion' and 'extraction' operators?
12. Why does the function to overload the 'subscript' operator return by reference?
13. What special precautions should be taken while overloading the 'subscript' operator for constant objects?
14. What are smart pointers? How are they created?
15. How are values of fundamental data types converted to class objects?
16. What ambiguity can arise in the following code? How can it be resolved?

```
class A
{
  public:
    A(int);
};

class B
{
  public:
    B(int);
};
```

```
void f(A);
void f(B);//function f() is overloaded

void g()
{
  f(1);
}
```

17. How can a class object be converted to a value of fundamental data type?
18. What are the two ways of converting an object of one class to an object of another? Describe the ambiguity that can arise if both methods are applied.
19. What is the advantage of using the new style casts over the old C-style casts?
20. Name the four new style casts provided by C++.
21. What is RTTI? What are its practical uses?
22. What is the difference between the `static_cast` and `dynamic_cast` operators?
23. What does the `const_cast` operator do? Which keyword of C++ can it be used instead of?
24. How can the `typeid` operator be used to find the type of a particular object?
25. State true or false.
    (i) New operators can be created by operator overloading.
    (ii) The `sizeof` operator cannot be overloaded.
    (iii) Number of arguments that an existing operator takes cannot be changed by operator overloading.
    (iv) Functions to overload the `new` and `delete` operators are always static.
    (v) The `dynamic_cast` operator throws an error if the type of the pointer that is passed to it does not match the type that is passed to it.
26. Modify the code given under the section on overloading the `new` operator to save memory when a large number of objects are created. Instead of having a union with the next pointer as a member, put another static data member that will count how many objects from the pool have had their addresses returned. When this counter becomes equal to the number of objects, another pool can be allocated. Compare the two codes for efficiency in memory usage.
27. Overload the 'equality' operator (==) for the class 'Distance'.
28. Overload the 'insertion' and 'extraction' operators for the class 'String'.
29. Overload the 'subscript' operator for the class 'String' so that it takes a character as a parameter and returns the position of its first occurrence. The output of the following code should be two.

```
String s1("abcd");
cout << s1['c'] << endl;
```

30. Overload the 'addition' operator for the class 'String' so that it adds two strings and returns the result. The output of the following piece of code should be 'abcxyz'.

```
String s1("abc"),s2("xyz"),s3;
s3 = s1 + s2;
cout << s3 << endl;
```

31. Overload the 'addition' operator for the class 'String' so that the output of the following code is 'c'. Introduce suitable checks for array bounds.

```
String s1("abcd");
cout << s1 + 2 << endl;
```

Moreover, the output of the following piece of code should be 'abxd'.

```
String s1("abcd");
s1 + 2 = 'x';
cout << s1 << endl;
```

32. Overload the bitwise exclusive OR operator (^) for the class 'Distance'. The overloading function should return true if the value of either of the two objects that are passed to the operator is not equal to zero. For the rest of the cases, the function should return false.

33. Refer to the section on overloading the 'pointer-to-member' operator. The operator has been overloaded so that objects of the class 'StrPtr' can mimic the behavior of pointers. In order to complete the picture, overload the dereferencing operator so that the following statements become equivalent ('p' is an object of the class 'StrPtr').

```
p->setString("abcd");
*p.setString("abcd");
```

34. Define two classes 'Polar' and 'Rectangle' to represent points in the polar and rectangle systems. Introduce a conversion operator function in class 'Polar' to convert its objects into objects of class 'Rectangle' and a conversion operator function in class 'Rectangle' to convert its objects into objects of class 'Polar'.

35. Consider the following class hierarchy:

```
class A
{
  public:
    virtual void f1() {}
};
class B : public A {};
class C : public B {}
```

In which of the following would the `dynamic_cast` operator return zero?

(a)
```
A * APtr = new C;
C * CPtr = dynamic_cast<C *>(APtr);
```

(b)
```
A * APtr = new B;
C * CPtr = dynamic_cast<C *>(APtr);
```

(c)
```
A * APtr = new C;
B * BPtr = dynamic_cast<B *>(APtr);
```

Chapter 9

Templates

OVERVIEW

This chapter explains the concept of generic programming using templates. Function templates along with their use and benefits are included. Class templates, their use and benefits are also included.

The Standard Template Library provides a number of useful class templates that can be used to meet various common programming needs. Important class templates of this library are also described in this chapter.

9.1 Introduction

We frequently come across functions that work in exactly the same way for different data types. Each of these functions has been designed to handle a specific data type. For different types of variables, only the keyword used to declare the variables upon which they work changes. The algorithm that these functions implement remains the same and therefore the structure of the function remains the same. One such function that immediately comes to mind is the one used to swap two values.

```
void swap(int & a,int & b)
{
  int temp;
  temp=a;
  a=b;
  b=temp;
}
```

Listing 9.1 A function to swap two integers

The preceding 'swap' function swaps the values of two integers. A 'swap' function that swaps two floats will have the following definition:

```
void swap(float & a,float & b)
{
  float temp;
  temp=a;
  a=b;
  b=temp;
}
```

Listing 9.2 A function to swap two float type numbers

We can notice that the two 'swap' functions are exactly alike except for the data type of the variables upon whom they work. It would be quite reasonable to expect that the C++ language provides us with a facility to write a common function that is independent of a data type but which embodies the common algorithm and that the C++ language on its own creates the actual function as and when the need arises. Having code at a common place has obvious advantages, namely ease in code development and ease in code maintenance.

This facility is provided in the form of templates. The programmer can create a template with some or all variables therein having unspecified data types. Whenever the template is invoked by passing arguments of a certain type, the C++ language on its own replaces the unspecified type with the type of the arguments passed. Such templates can be created for individual functions as well as entire classes.

9.2 Function Templates

The syntax for creating a template for a generic function is as follows:

```
template <class T, ...>
return_type function_name(T arg1, ...)
{
  //statements
}
```

Listing 9.3 Syntax for a function template

The template definition begins with the `template` keyword. This is followed by a list of generic data types in angular brackets. Each generic type is prefixed with the `class` keyword and, if the template function works on more than one generic type, commas separate them. Thereafter, the function template is defined just like an ordinary function. The return type comes first. This is followed by the name of the function, which in turn is followed by a pair of parentheses enclosing the list of formal arguments the function takes. However, there should be at least one formal argument of each one of the generic types mentioned within the angular brackets.

For example, the template for the function 'swap' can be as follows:

```
/*Beginning of swap.h*/
template <class T>
void swap(T & a, T & b)
{
  T temp;
  temp=a;
  a=b;
  b=temp;
}
/*End of swap.h*/
```

Listing 9.4 Template for the function 'swap'

Now, suppose the function 'swap' is called by passing two integers. The compiler generates an actual definition for the function by replacing each occurrence of T by the keyword `int`.

```
/*Beginning of swap01.cpp*/
#include<iostream.h>
#include"swap.h"
void main()
{
  int x,y;
  x=10;
  y=20;
  cout<<"Before swapping\n";
  cout<<"x="<<x<<" y="<<y<<endl;
  swap(x,y);    //compiler generates swap(int&, int&); and
                //resolves the call
  cout<<"After swapping\n";
  cout<<"x="<<x<<" y="<<y<<endl;
}
/*End of swap01.cpp*/
```

Output

```
Before swapping
x=10 y=20
After swapping
x=20 y=10
```

Listing 9.5 Calling the template for the function 'swap' by passing integers

Similarly, if the function 'swap' is called by passing two floats, the compiler generates an actual definition for the function by replacing each occurrence of T by the keyword `float` and so on.

```
/*Beginning of swap02.cpp*/
#include<iostream.h>
#include"swap.h"
void main()
{
  float x,y;
  x=1.1;
  y=2.2;
```

```
cout<<"Before swapping\n";
cout<<"x="<<x<<" y="<<y<<endl;
swap(x,y);    //compiler generates swap(float&, float&);
              //and resolves the call
cout<<"After swapping\n";
cout<<"x="<<x<<" y="<<y<<endl;
}
/*End of swap02.cpp*/
```

Output

```
Before swapping
x=1.1 y=2.2
After swapping
x=2.2 y=1.1
```

Listing 9.6 Calling the template for the function 'swap' by passing floats

Objects of classes can also be passed to the function 'swap'. The compiler will generate an actual definition by replacing each occurrence of T by the name of the corresponding class.

```
/*Beginning of swap03.cpp*/
#include<iostream.h>
#include"swap.h"
#include"Distance.h"
void main()
{
  Distance d1(1,1.1),d2(2,2.2);
  cout<<"Before swapping\n";
  cout<<"d1="<<d1.getFeet()<<"'-"<<d1.getInches()<<"''\n";
  cout<<"d2="<<d2.getFeet()<<"'-"<<d2.getInches()<<"''\n";
  swap(d1,d2);  //compiler generates swap(Distance&,
                //Distance&); and resolves the call
  cout<<"After swapping\n";
  cout<<"d1="<<d1.getFeet()<<"'-"<<d1.getInches()<<"''\n";
  cout<<"d2="<<d2.getFeet()<<"'-"<<d2.getInches()<<"''\n";
}
/*End of swap03.cpp*/
```

Output

```
Before swapping
d1=1'-1.1"
d2=2'-2.2"
```