CHAPTER 6

Operator Overloading

In the previous chapter, we have discussed class or object creation. Manipulations on class objects were accomplished be sending messages in the form of member function calls to objects. But then with mathematical classes, the member function call syntax is complex. The C++'s operator set could be exploited using the concept of operator overloading. Operator overloading is the process of assigning new meaning(s) or operations to operators from the C++ operator set. It is referred to as overloading because more than one meaning is associated or assigned for operator or new meaning(s) or operations in addition to the existing meaning are assigned for the operator.(or) in other words, operators are overloaded to perform different operations as per the requirements.

One simple example for an already overloaded operator in the language is << and >> operator. The first operation is the bitwise left shift and right shift operations. The other overloaded operations associated with the above operator are stream insertion or extraction [output–input statement] in C++. Another classical example of overloading is the binary + operator, since the + operator performs overloaded operations of pointer, integer and floating number arithmetic.

Depending on the context in which it is used overloaded operators perform the intended or correct operations. Almost (except of a few that shall be mentioned) all the operators of C++ operator set can be overloaded. In this chapter, we shall explore the implementation details of operator overloading. But then operator overloading can always be substituted by the member function syntax, but then it is one solution wherein the appropriate member function call is performed automatically by the compiler, not expecting the programmer or end user to explicitly call the appropriate function. It is just a more realistic and convenient way of manipulating class objects.

6.1 Introduction

Operators that are overloaded or assigned meaning by the programmer already have a meaning or operation associated with them. When we stated that overloaded operators perform the appropriate operation depending on the context in which they are used, it is nothing but the operand type. Thus, when an overloaded operator is made use of or applied with built-in data types, the existing (normal) meaning of the operator is in force. When used with objects or user-defined data types, the newly assigned or overloaded meaning is in force. Operator overloading though a powerful programming technique should be used carefully. An example

of operator overloading would be to define the + operator to perform addition of matrices [assuming we have a user-defined data type class Matrix].

Operator overloading is achieved or implemented by defining a function with the keyword operator followed by the operator that is to be overloaded [in fact, this is the name of the function or the function identifier] and within the function definition or body of the function the logic associated with the operator's new function or meaning is specified. To overload the operator +, the function header would be of the form operator +. Operator overloading can be implemented via both member functions as well as non-member functions. The choice between a member function and a non-member function overload depends on the operator's application. We shall provide details on this choice in the next section.

6.2 Rules to be Followed in Operator Overloading

6.2.1 Overloaded Meaning should be Realistic

Operator overloading should be intuitive and realistic. That is to say that though it is always possible to syntactically overload the + operator to perform subtraction on user-defined data types, such an overload that is unrealistic and confusing. C++, for that matter any OO language should support reusability and extensibility. Programmers who are to reuse or extend or maintain such code would proceed with the natural or normal conventions and would require certain effort to first place spot the meaning or operation associated with the respective operator. To put it in simple words, such overloading is unscrupulous and idiotic.

6.2.2 Built-in Operator Overloading Support

Two operators that can be used with both built-in and user-defined data types [class objects] with no overloading required are the = (assignment) and the & (address of) operator. These two operators when applied with built-in data types or used-defined data types perform the operations of assignment and returning the address of the [built-in or user-defined data type] in memory. As we have already mentioned, operator overloading is more often required for classes representing mathematical entities such as complex or rational numbers, wherein it would be extremely realistic or convenient to overload the arithmetic or relational operators on such user-defined data types. And in such cases, the requirement would be to overload quite a few or substantial number of frequently used operators.

6.2.3 User Definition Support

Operator overloading mechanism is to provide a precise or concise notation for manipulation of user-defined data types. Operator overloading similar to friend functions is neither granted nor automatic. It has to be explicitly declared and defined by the programmer. The programmer must explicitly define the operator functions to perform the required operations. A good programming practice is to overload operators on user-defined data types, to perform operations that are quite similar to operations on built-in data types. As a part of our discussion on friends, we have mentioned that friends feature of C++ shall be used in operator overloading.

6.2.4 Methods and Precedence of Overloading

Operator overload functions can either be member functions of user-defined data types or friend functions or non-member, non-friend functions. Most of C++ operator can be overloaded, the exception's being `*,::,?:` and size of operator. Attempting to overload such non-overloadable operators cause syntax error. The other dont's of operator overloading are as follows. The precedence of an operator [language-dependent] cannot be changed by operator overloading, since allowing it would cause the existing or fixed precedence vs. programmer-defined precedence determination is complex. An easy way out from this precedence bottleneck is to exploit the parenthesizing to change the order of evaluation of overloaded operators.

6.2.5 Associativity and Arity cannot be Changed

The associativity of operators cannot be changed by overloading. The number of operands which the operator takes or operates on cannot be changed by overloading. This is also referred to as arity of operators, the naming from the existing convention of unary, binary and ternary operators. Thus, the arity of operators cannot be changed via operator overloading. Thus, unary or binary operators can be overloaded only as unary or binary operators. The ternary operator `(?:)` cannot be overloaded. Certain operators that have both unary and binary operations associated with them such as `&` → address of (unary) and bitwise, and (binary) and `*` → dereferencing (unary), and multiplication (binary) can be overloaded separately.

6.2.6 New Operators cannot be Created

One syntactic compulsion with operator overloading is that programmers can only overload existing operators [operators defined in the language set]. The programmer cannot altogether create new operators, only existing operators can be overloaded. Thus, assuming that one can create operators such as `:=` or `**` is invalid. Operators that do not belong to the C++'s operator set cannot be overloaded. Attempts to create new operators via operator overloading will cause syntax errors. The compiler simply does not recognize such new operators. Thus, via operator overloading it is only new meanings for existing operators and not new operators that can be created. And also the context or meaning of how an operator works on objects of built-in types cannot be changed by operator overloading.

The control lies with the programmer only for manipulations on objects of user-defined classes. With built-in data types or objects, it is always the predefined meaning that is applicable. Operator overloading always works only on or with objects of user-defined types or with a mixture of user-defined and built-in-type. But at least one operand of an overloaded operator must be of user-defined type for the overloaded meaning to be in force. When all the operands of an overloaded operator is of built-in or existing (fixed) meaning of operator that is applicable or in force. Thus, the powerful programming feature of operator overloading wherein programmer has control over meaning or operations of operators lies only with user-defined data types.

6.2.7 Operator Overloading should be Explicit

Operator overloading should be always explicitly specified by the programmer. There is no implicit overloading performed by the compiler on the programmers behalf. Assuming that

shorthand notation (such as +=) operations are overloaded when their basic forms [+,=] are overloaded is a syntax error. That is, if + & = operators are overloaded individually += is not overloaded implicitly and if required should be overloaded explicitly by the programmer. Another example for implicit overloading not possible is that related operators do not get overloaded implicitly. That is assuming that on overloading ==, != is also overloaded is a syntax error. All operators need to be explicitly overloaded by the programmer. But then already overloaded operators should be utilized to the extent possible. That is, if + and = have been overloaded by the programmer, they should be used while overloading +=. Code repetition should be avoided always, since they are cumbersome in terms of both code size and program maintenance.

6.3 Methods of Defining Operator Functions

In the earlier section, we have mentioned that operator overloading functions can either be member functions or non-member functions, or friend functions. In case of non-member function overload, all the arguments (mostly two, with binary operators in mind) must be explicitly passed to the function, since it being a non-member function would require information about all the values (operands to be precise). In the case of member function overload, only one argument should be passed. Since it is a member function, the objects with which it gets called or associated is already known and hence for binary operator overload via member function only one operand is required.

6.3.1 Member Function Overload

For better understanding overloading of binary operators like `a+b` [assuming `a` and `b` are user-defined class objects] would be parsed as `a.+(b)`. Thus, the object with which it gets invoked is actually the LHS operand of binary operators. Hence only one operand is to be passed to the function call explicitly; the other operand is implicitly taken by the compiler. In fact, the compiler uses `this` pointer to obtain the operand. `This` pointer is nothing but the object for which a member function was executed most recently, nothing but the object or operand that was not passed as a part of the function call. That is for the expression `a + b` [again `a` and `b` are assumed to be instances of user-defined types], we already stated the parsing would be of the form `a.+(b)`. Thus, within the member function [operator `overload function→ operator + (b)` referring to the this pointer is actually referring to the address of object `a`.

6.3.2 Non-member Function Overload

The operator function mentioned within braces above might not be entire in its signature. It is from the number of operands required point of view that operator overloading is explored in this section. We shall be complete with the signatures of operator overload functions in the programming examples that are to follow. Thus, the LHS operand of a binary operator must be always an object of the class of which it is scheduled to be a member function. If the LHS operand will be an object of a different type [but not an instance of the class to which it can be a member function], the operator overload function should always be a non-member function. And if such non-member functions require access over private data members of another class,

they are better off to be declared as friend functions of such classes whose private members may be referred to within this non-member operator overload function.

A classical example for a friend non-member function overload would be the overloading of `<<, >>` operators with respect to user-defined class objects. Let us assume we have a user-defined class called `Integer_Array`. A statement of the form `Integer_Array o1 (10)` would create an integer array of size 10, initialized to `0's`. Assuming that the array has been assigned values [some other member function available], the programmer at some stage or the other would require to display the array contents. It would be convenient if a redirection or output statement of the form `cout << o1` to display all the array contents [in this case `o1`] is available. Rather than the end user of this class having to go through each index of the array, this indexing logic can be specified as a part of the operator overloading logic for the `<<` operator. In other words, the end user statement of the form `cout << o1` would be interpreted by the compiler as the new overloaded meaning of the `<<` operator to redirect the entire objects contents on to the screen.

Readers should observe that in a statement of this form `[cout << o1]`, the LHS operand of the operator `<<` is always an instance of the built-in class ostream [`cout` is an object of class `ostream`]. Thus, such operators should always be overloaded as non-member functions. And also since the redirection operation would often require access over private data members [the array contents], the `<<` operator is normally overloaded as non-member friend function of the user-defined class `Integer_Array`.

In the next section, we shall get into the programming intricacies or niceties of these `<<` and `>>` operators overload. As is obvious the `>>` operator would also be overloaded as non-member friend function since LHS operand of `>>` will always be of type built-in class `istream.[cin]` and again would require access to private data members of the user-defined class `Integer_Array`. The last form of operator overload, non-member, non-friend function is as well possible but then such a function to access the private data members of a class [user-defined] with which the operator is applied, public interface functions should be available in the class which should perform this job. Thus having discussed on member vs. non-member operator overload function issues, for better understanding, let us go though this example.

A non-member function overloaded operator such as `cout << userdefinedobj;` is parsed by the compiler (or leads a call to) operator `<<(cout,userdefinedobj)`. A member function overloaded operator such as `+` when used as follows `a + b` is parsed by the compiler as `a.operator + (b)`. In both the above examples it is assumed that corresponding operator functions have been appropriately defined. Thus, operator member functions of a user-defined class are called only when the left operand of a binary operator is always an object of that class or when the single operand of a unary operator is an object of that class.

One valid reason to go in for a non-member function overload is to make the operator support commutative property. For better understanding of this, let us assume `o1` and `o2` are instances of two user-defined classes A and B. If `o1` and `o2` are to be added [assuming A and B are representing mathematical entities], the statement issued will be of the form `o1 + o2`. Let us assume that `+` has been overloaded in class A as member function in which case the compiler parses the above expression as `o1.operator +(o2)`. No problems as of now.

With normal arithmetic `+` is commutative and based on the principle that operator overloading will be to the extent possible identical to normal or fixed operations, the `+` operation on objects of `o1` and `o2` should be commutative. That is a statement of the form `o2 + o1` should be valid. But if `+` is overloaded as a member function of class A, then the LHS operand

should always be of type class A, whereas in this case or for + to be commutative, the LHS operand can also be an in stance of class B. Thus, at least operators that need to support commutative property should be overloaded as non-member functions. For example, in the above case if + is overloaded as a non-member function (can also be a friend function), the statements(s) `o1+o2 or o2 + o1` get parsed as follows:

`operator + (o1,o2)`→ first case and operator `+ (o2,o1)`→ second case

In all the signatures that we have seen with respect to this chapter on operator overloading, we have not expressed concern as to how passing arguments [objects, which are operands to overloaded operators] by value or by reference. This was with purpose, to understand the issues involved in operator overloading in a comprehensive manner. The above sections were devoted to operator overloading. It so happens that operator overloading is implemented using a function with the keyword `operator`. We did not want to confuse readers with issues of passing by reference of by value or by constant reference, at least while discussing operator overloading. The programming examples to follow, we will mostly pass objects [which are nothing but operands to the overloaded operator] by reference, if modification or update is required and as constant reference if only read permissions are required over the object or operand.

Note: The reasoning behind the LHS operand of a binary operator to be always of class type where the operator is overloaded as a member function is that the compiler, for all member function overloads, associates the operator function call with the LHS object or operand. Thus, if a binary operator that has been overloaded as a member function of one user-defined class and is applied with an object of built-in or some other user-defined class type on the LHS of the operator, the compiler [after having searched for a non-member function version of the overloaded operator] would associate with which it gets associated on the LHS, where it would not be able to locate one such function. Thus this is the reasoning behind the syntactic restriction that LHS operand of a member function overloaded operator is always of the respective class type. This is a syntactic restriction and violations if are resolved at compilation stage itself. We have had enough on issues involved in operator overloading. It's high time we get exposed to programming examples involving operator overloading. The following sections will elaborate on overloading of various operators from C++ operator set.

6.4 Overloading of << and >> Operators

Let us assume a student information [details of Name, Register Number and Age] is to be created. We will create a class called Student whose data members are Name, Register Number and Age. We shall overload the stream insertion (<<) and stream extraction (>>) operators for input or output of student information at one shot. [`eg:cout<<student1; cin>>student1`] should either accept or redirect values for all three members values to the screen] obviously the LHS operand being an instance of a predefined class [istream or ostream] we shall go in for a non-member friend function overload since both operators require access over private data members of the class. The code is as shown in Figure 6.1.

```
1.  class student
2.  {
3.  friend ostream & operator << (ostream &, const student &);
4.  friend istream & operator >>(istream &, student &);
5.  private : char name [50];
6.  char regno[30];
7.  int age;
8.  };
9.  ostream & operator<<(ostream & o, const student & stud)
10. {
11. o <<"Name of Student is \n"<<name<<endl<<"Register
    No is \n"<<regno<<endl<<"Age is"<<age<<endl;
12. return o;
13. }
14. istream & operator >> (istream & i, student & stud)
15. {
16. i >> "overloaded Input Operator";
17. i >> stud.name;
18. i >> stud.regno;
19. i >> stud.age;
20. return i;
21. }
22. int main ()
23. {
24. Student S1;
25. cout <<"Enter student details \n";
26. cin >> S1;
27. cout <<"Displaying student details \n";
28. cout >> S1;
29. return 0;
30. }
```

Figure 6.1: C++ code to overload input and output operators.

6.5 Code Interpretation

The student class has two non-member friend functions, operator << and operator >> each of which accepts the respective stream reference as an argument, because they are overloaded as non-member functions. The output operator function accepts the student object as constant reference, since it is only a read operation and read access is what is required over the student object. The input operator accepts the student object as a non-constant reference, since it being an input operation, one would require write permissions over the object. In both cases, arguments are passed by reference to avoid the duplicating overhead associated with pass by value mechanism. Now let us get on with the definition of the operator overload functions.

The named `ostream` reference is used within the definition of the output operator definition and each of the three data members of the student class are redirected to this named ostream reference (o) separately. The definition of the input operator overload function is quite similar to the output operator definition. In the driver function, we test drive the overloaded operators by accepting and redirecting values for/of the student object at one shot. When the compiler encounters a statement of the form `cout << stud`; in main the compiler generates the function call of the form operator `<< (cout,std)`, wherein the formal parameters o and `stud` become

aliases for cout and student objects. Both the operator functions return a reference to the respective stream to support cascaded input or output operations. Cascading of operators has been explained in the earlier chapter. Cascading of operators avoid duplication of the object names associated with the input or output operation. At end of an operator function call, the returned `ostream` or `istream` reference is used to process the remaining input or an output operation that needs to be completed. The definition of the input operator is similar to the output operator, except for the input `stream` reference replacing the `ostream` reference.

In this example, we overload the `stream` insertion and extraction operators to output or input student details of Name, Regno and Age at once. Name, Regno and age are private data members of the student's class. C++ allows insertion and extraction of character arrays at once. Thus, we will not go though each index of the character arrays to redirect the array contents or accept input for the array. Well, in this class we have purposely excluded the constructor and destructor definitions. It is left as an exercise for the reader and does not have any significant role to play in operator overloading. The two operators to be overloaded << and >> are defined as non-member friend functions for reasons explained already. Now elaborating on their signatures which are as follows:

```
friend ostream & operator << (ostream &, const stud &);
friend istream & operator >> (istream &, stud &);
```

The keyword `operator` denotes that it is an operator overloading function. The function names are << and >> respectively. The keyword `friend` indicates that the two functions (non-member) will be friends to the class student and thus have access to the class's private data members. Since << and >> are both binary operators and are operator-overloaded as non-member functions, we need to pass both the LHS and RHS arguments (operands) of the << and >> binary operands of the << and >> binary operators. In the stream insertion, the LHS operand is always an object of type class `ostream` and the RHS operand is always an object of user-defined type class student.

Thus, the two arguments passed to `friend` function operator << are of type `ostream` and student classes. They are passed by reference to avoid the duplicating or copying overhead associated with call by value and also references to the formal parameters of output and `stud` should actually refer to the original `cout` and user-defined object of class student. Observe that the output operation requires only read permissions over the student object passed by reference. Hence we make it a constant reference to enforce the principle of least privilege. The return type of the function is a reference to `ostream`. This is with purpose. We observe that the existing stream insertion and extraction operators support cascading. That is multiple redirections or indirection can be supported with a single `cout` or `cin` statement by cascading or chaining the << and >> operators as follows:

Let us assume `a` and `b` are of type integer. Thus, to redirect `a` and `b` at one shot using a single `cout` statement and cascading of <<operator, we issue the statement `cout << a << "\n" <<b`.

The cascaded indirection is quiet similar to the above redirection operation. To remember a fact, in our discussion on `this` pointers, we have mentioned that one if its main application is in supporting cascading of operators, the two operator non-member functions of our student class return references to `ostream` and `istream` precisely to support cascading. Within the respective definitions of the non-member functions << and >>, we return the formal reference parameters of output and input which is nothing but aliases for `count` and `cin`. Thus, when

a statement such as `cout << stud1 << stud2` is issued in the test driver function, assuming that `stud1` and `stud2` are two objects of user-defined class student, the compiler parses the above statement as operator `<< (cout,stud);` at the end of this non-member function call returns a reference to `ostream[cout alias o]`. Thus, the remaining statement is treated by the compiler as `cout<< stud2` which is again parsed as operator `<< (cout,stud2)`, at which point the returned `ostream` is captured by output buffer.

Cascading of >> (stream extract input) operator is quiet similar to the << operator except that the predefined class is `istream` and the function accepts references to class `istream` and student and returns references to `istream`. The reasoning behind two references arguments being passed and a reference being returned is exactly the same as that explained for `stream` insertion operator << overload. The definitions of the two non-member functions operator overload are quite simple. The `stream` insertion operator overload definition specifiers the logic for redirecting the three data members of the class student. Note that C++ allows `char` arrays to be redirected at one shot. The function after displaying each data member returns a reference to `ostream` variable (o — which is actually an alias for `cout`).

The stream extraction operator >> function defines the logic for accepting the values for three data members. At the end of its processing, the function returns a reference to `istream` (i) which is an alias for `cin` object. Note that the student object passed by reference to the operator >> function is non-constant. If this is because the function of >> would be to accept values from the keyboard and assign it to the respective data members which means modification or assignment operation will be performed over the object (student object) values will be read in from the keyboard and assigned to the `studentobject.name, studentobject.regno and studentobject.age`. Thus, write permissions are required over the object. Hence the student object reference that is passed to the operator >> non-member function is passed as non-constant reference argument.

Well, again since cascading of >> is supported with the existing meaning of >> operator, the overloaded version of >> should also support cascading. That is as we overloaded the << operator to support cascading or redirection of many student objects at one shot, so should the >> operator also support cascading or in other words details of more than one student object should be possible to read in using a single `cin` and cascaded or chain of >> operators. Thus the operator function operator >> is scheduled to return a reference to `istream` [the formal parameter being input which is in fact an alias for `cin`]. Within the definition of this non-member function, the logic for accepting values for individual data members is specified. Again we exploit the fact that C++ allows character arrays to be read in at one shot.

At the end of its processing, the function returns the `istream` reference [formal parameter input is returned] to enable cascading operation. Thus, the statement `cin >> stud1 >> stud2`; assuming `stud1` and `stud2` are objects of the user-defined class student, the compiler parses the above expression as operator `>> (cin,stud1)`; at the end of which a reference to `istream` (input—alias for `cin`) is returned. The remaining portion of the statement is then parsed as operator `>> (cin, stud2)`. The `main()` function serves as the driver function to test drive out student class. As is obvious the function creates student objects `stud1` and `stud2`. It then makes use of the programmer overloaded >> operator to read in values for data members of the objects `stud1` and `stud2`. Then we check if values were properly read in or not, we make use of the overloaded << operator to redirect the data members of `stud1` and `stud2` or state of `stud1` and `stud2` onto the screen. In the next section, we shall overload more complex operators.

6.6 Overloading an Array Class

In this section we will discuss on a programming example to overload a user-defined class `Array`. As it is the language already has a built-in array declaration supported. Thus, when we start out constructing a new array data type, there should be some additional features made available with this creation in comparison with the already available construct. To design our class in an efficient manner, let us first go though the bottlenecks associated with the existing array construct. The various problems associated with the array construct supported in the language are explained below.

Arrays can be walked off at either ends or what is often referred to as Array out of Bounds is a common problem with the built-in array construct. We will overload the `[]` array access operator to check for the out of bounds [exceeding the lower limit [index being `< 0`) and upper limit [index being `>` maximum size of the array] condition and then suitably perform the requisite action. Only character arrays can be input or output at once without having to overload the `<<` and `>>` operators. In our `IntArray` class, we shall overload the stream insertion and extraction operators to perform input and output of even integer arrays at once.

Arrays cannot be compared with equality of relational operators with when using the built-in array construct of the language. This comparison is not allowed with the existing construct since array names are nothing but pointers to the starting address of the array in memory or what is also referred to as base address of the array. In the `IntArray` class we create, we shall overload the `==` and `!=` operators to check if two Integer arrays are equal in their contents or not.

Also arrays cannot be assigned to another array identifier using an assignment statement with the built-in array constructs. We will overload the assignment operator to allow assignment of one integer array object to another integer array object. This assignment operation is not allowed with the built-in construct since array names are constant pointers and a constant pointer cannot appear on the LHS of an assignment operator.

Most often when arrays are passed as arguments to functions, their sizes are as well passed. But then with the integer `Array` class that we shall develop, manipulation of integer array objects via functions will not require the sizes of the array to be passed as an argument to functions that manipulate such objects. Instead the size of the array objects is made available as a data member and hence there is no requirement to pass the array size of functions that manipulate such array objects.

6.6.1 IntArray Class

The `IntArray` class has as its data member's size [size of the array], an integer pointer `[aptr]` that points to the starting or base address of the array. Since the programmer would like to keep a count of the total number of `IntArray` objects created at any point of time, we declared a static variable array count of type integer. It is declared as a static member, so that each time an object of class `IntArray` is created, the count variable is incremented by one. If this is declared as a normal data member that would be created on a per object basis, then the array count variable will be created separately for each object and thus the increment on such a member will not be a cumulative one, as each object has a separate copy of the count variable. Now let us concentrate on the various functions that this class is to support. First, let us complete the constructor for the class.

6.6.2 Constructor

The signature of the constructor is as follows:

```
IntAray (int =20);
```

The integer argument passed to the constructor is the size of the array. In case the user fails to specify that the array size at the point of object creation, then the array size is defaulted to 20. Thus, the above signature serves both as the default as well user-defined constructors. Definition of the user-defined constructor is as shown below.

```
IntArray(int sz)
{
size = sz;
aptr = new int [size];
++ arraycount;
for (int i=0;i<size; i++)
aptr[i]=0;
}
```

The passed size value (`sz`) is assigned or set to the size data member. Readers should note that the memory required to store the array elements of the required size for the array variable or pointer has not yet been allocated. Well, we wanted to create arrays of the required size dynamically at run time and not statically at compile time in which case the size of the array cannot exceed the specified maximum limit. And also as has been already mentioned statically allocated arrays may cause wastage of storage space the array size is always greater than the maximum value or limit specified in the source code. Thus, to avoid the above-mentioned bottlenecks, we decide to allocate space for the array dynamically as per the required size. Now having fixed or set the size of the data member, the next job of the constructor would be to allocate the memory for the required size. The statement `aptr = new int [size];` allocates memory of the required size and returns a pointer to `aptr` which is the base address of the array. Having allocated the memory of the required size, we then initialize the array contents to 0 using a `for loop` as follows:

```
for (int i=0;i<size; i++) aptr[i]=0;
```

6.6.3 Destructor

```
Signature is ~IntArray();
Definition:
~IntArray()
{
delete [ ] aptr;
};
```

The destructor deallocates the space allocated for the array using the statement `delete[ ] aptr`. Remember it is a collection or an array of spaces that is to be deallocated and hence the `[]` within the delete `aptr` statement is applied.

6.6.4 Assignment Operator Overload(=)

Now we will overload the assignment operator to assign array objects to one another. Note that both the LHS and RHS operands of the binary operators (=) are of the same user-defined (`IntArray`) type. Hence we will overload this operator as a member function. Since it will be overloaded as a member function, we will pass only one argument, the RHS operand (also of `IntArray` type) to this function. Since we require only read permissions over the RHS operand [array objects whose contents should be read and assigned to the LHS array objects], we shall pass it as a constant reference. We pass it by reference to avoid the duplication overhead, this reasoning remains valid for the other operators to be overloaded, as well. Thus the signature for the = operator.

Note that the existing = operator supports cascading, that is a statement with normal variables of the form `a = b = c` is executed right to left, that is c's contents are assigned to `b` which is then assigned to `a`. When this is the case with the normal meaning of the operator, based on the principle that operator overloading should be intuitive and similar to the existing meaning of the operator. That is cascading should be supported even in our overloaded version of =. That is multiple array objects assignment in a single statement should be possible.

For this, the function returns a reference of the array type. The reasoning behind this return type is the compiler would parse an overloaded = application in an expression of the form `o1=o2=o3`, where `o1,o2` and `o3` are assumed to be objects of class `IntArray`, right to left as follows:

First, the part `o2=o3` is parsed as `o2.=(o3)`. Then the remaining expression `o1. = (o2=o3)` would be parsed as `o1. operator = (o2)` if and only if the call `o2.=(o3)` returns an operand of the `IntArray` type. The returned reference is also made a constant reference to prevent multiple assignments in a cascaded application of the = operator. This is because parenthesizing the above expression can change the right to left normal evaluation order. By returning a constant reference (which will not be modifiable after only one assignment (initial)), we prevent manipulated multiple assignments such as `((o1=o2)=o3)`. Thus the signature of = operator is as follows:

```
const IntArray & operator = (const IntArray &);
Now let us go though the definition of the = operator function.
const IntArray & operator = (const IntArray & rt)
{
if (&right != this)
{
if(size !=rt.size)
{
delete [] aptr;
size = rt.size;
aptr = new int [size];
}
for (int i=0;i<size;i++)
aptr [i]= rt.aptr[i];
}
else
return * this;
}
```

Before performing assignment operation we need to check if we are trying to perform self-assignment. That is assigning the same object to itself which is meaningless and would drain system resources unnecessarily. This self-assignment test is performed by `(if (&right != this))`; `&right` returns the address of the passed or right-hand side argument and this actually refers to the address of the object or LHS operand since `this` pointer refers to the object with respect to which a member function was invoked most recently. Thus, the above statement checks if the address of the LHS and RHS operands or objects are the same or not. If and only if the objects differ in their addresses do we proceed with the assignment operation else we proceed to the next cascaded operation if any.

Now proceeding with the assignment logic, the first thing we need to check is if the size of the LHS array object is equal to the RHS array object size. For an array to be assigned to another array the first requirement would be that the size of the LHS array be the same as that of RHS array. If not the size of the LHS array should be resized to the size of the RHS array. This, is precisely what is performed by the inner if block. Now if the LHS array should be resized, then the memory allocations for LHS array should be resized. Thus, when the sizes of LHS and RHS array objects are not the same, we reclaim the space allocated for LHS array `[delete [ ] aptr];`, then allocates space for the resized array `[aptr = new int [size]]`. Now having resized (if required), we are left with the task of assigning RHS array values to the LHS array. This is performed with the following statements:

```
for (int i=0;i < size; i++)
aptr [ i ]= rt.aptr[i];
```

Finally, after having assigned (if this is not self-assignment) or skipped self-assignment operation, we return the `this` pointer or the address of the object with respect to which the member function is invoked. This return is to support cascading. Note that reference to size or `aptr` within a member function is actually the LHS array object's size or `aptr`. This is so because the compiler automatically associates the member function call with the operand (array object) on the LHS of the operator being overloaded.

6.6.5 Comparison Operator Overload(==)

Now we shall overload the == operator to compare if two array contents are equal to one another or not. Again this is overloaded as a member function for similar reasons explained in the earlier overload. Since it is only read permission over the RHS operand or object and read operations that will be performed within the comparison function, we make both the RHS operand as well as the function constant. C++ supports an additional data type called Boolean (keyword `bool`) to indicate the values of True or False. Hence the return type of the function is `bool` [Boolean]. The equality operator in its normal or existent meaning does not support cascading. Hence, our overloaded version of == will not support cascading and also the chance of such an application is extremely rare. Thus, the signature for overloading the == operator in our `IntArray` class `is:bool operator == (const IntArray &) const`.

Now getting on with the definition, arrays can be equal if and only if first place their sizes are the same. Thus, we check if the array objects on LHS and RHS of the == operator are same in size or not. If their sizes are different, there is no point in proceeding to check if their contents match or not. Thus we simply return FALSE and stop checking for array equality.

Now, if array sizes are equal, then we proceed to the logic of matching the array contents. Even if at one index the arrays differ in their values, we return FALSE and stop further equality checking. The function returns TRUE if and only if the values at all the array indexes [limit—user-defined at run time] are equal. The definition for the == operator is as follows:

```
bool operator == (const IntArray & r) const
{
if (size !  = r.size)
return FALSE;
for (int i=0;i<size;i++)
if (aptr[i] !=r.aptr[i])
return FALSE;
return TRUE;
}
```

6.6.6 Overloading the != Operator

Having overloaded the == operator, now we shall overload the != operator based on the principle that operator overloading should make use of already overloaded operators to the extent possible, we shall use the already overloaded == operator to overload the != operator. The result of a != operator will also be of Boolean type, with possible results of TRUE/FALSE. For better understanding let us assume a and b to be two integer variables with values of 4 and 4. Thus the test if (a ==b) returns true. Now, to apply the != check all we need to do is if (!(a==b)), in this case both a and b being equal the result of a==b will be TRUE and ! (TRUE) will be FALSE. In fact, the values are equal. Now, if a =4 and b =3 a==b would return FALSE and ! (FALSE) would return TRUE, to indicates that a and b are not equal to one another. Now, let us get back to our IntArray class overloading example. The signature of != operator is as follows: `bool operator != (const IntArray &) const;`

The definition is a simple one line statement as follows:

```
bool operator != (const IntArray &) const
{
return !  ( *this == right);
}
```

Yes, the single line return statement solves the task. The conditional check * this == right actually checks for equality of * this [LHS object] and RHS object. Note that * this actually dereferences the base address of the LHS array object, since this pointer refers to the address of the object for which the member function is executed off late. The remaining logic is quiet similar to as that was discussed for the simple case of a & b variable equality checking.

6.6.7 Subscription Operator Overload ([])

During overloading of array subscription operator [], the operator is overloaded to include the out of bounds checking. The operator is overloaded as a member function because the LHS operand is always an instance of the user-defined class IntArray. The main task of

the operator function is to test if the index used with the array is valid or not. That is not exceeding the maximum size of the array or the index being negative.

Thus, the operator [] function accepts the index or subscript as an argument. There is another issue to be addressed, the return type. Now, the indexing `[a[i]` can be both on the LHS or RHS of an assignment, that is `[I = a[i] or a[i] = c]` or one step further `a[i]=c[i]`. Thus, in one case the result (return type) of the operator should create an `lvalue` `[a[i]=c]` and in the other case should create an `rvalue [c=a[i]]`. `Lvalue` is the memory location or address where the array value `a[i]` is stored or will be stored. `Rvalue` is the content at a memory location or the value of `a[i]`. The first case where an `lvalue` should be created the signature looks as follows:

```
int & operator [ ] (int);.
```

The second case where an `rvalue` should be created the signature of [] looks as follows:

```
Const int & operator [ ] (int) const;
```

The `const int` reference returned ensures that only read [`rvalue` creation] and not write [`lvalue` creation] operation is allowed. The function is also made a constant function to allow constant array object call the function. The definitions of the above two operator overload functions are similar. We check for the index to be within the valid range or not. If yes then either the address or the value is returned. Else the index is forcibly set to the valid minimum or maximum index. Well, this is one solution to overcome out of bounds error. Another solution could be to terminate the program in execution by calling abort. The definition of the operator [] function is as follows:

```
int & operator [] (int sscript)
{
if (sscript >=0 && sscript < size)
return aptr[sscript];
else
abort();
}
const int operator [ ] (int sscript) const
{
if (sscript >=0 && sscript < size)
return aptr[sscript];
else
abort();
}
```

6.6.8 Input–Output Operator Overload (<<, >>)

Overloading the stream insertion or extraction operator to allow integer arrays to be input or output at one shot is quite simple. The signatures of the << and >> operators are as follows:

```
friend ostream & operator << (ostream &, const IntArray &);
friend istream & operator >> (istream &, IntArray &);
```

The earlier example was devoted for << and >> overloading. The reasoning behind both operators being overloaded as friend non-member function is similar to that in the previous example. Let us get on with the operator function definitions. The logic would be to traverse through each index of the array via a `for` loop and make use of the existing << and >> to accept or redirect one array value. The function definition is as follows:

```
ostream & operator << (ostream & o,const IntArray & a)
{
for (int i=0;i<a.size;i++)
o << a.aptr[i];
return o;
}
```

6.6.9 Copy Constructors

Well, in our earlier discussions on constructors we have seen two kinds of constructors, namely the default and user-defined constructors. In fact, there are two more constructors, which are called copy and conversion constructors. In this section, we shall provide details on these two constructor types. Well, in the previous example we overloaded the = assignment operator to allow assignment of objects to one another. With this = operator, the objects are assumed to be declared. But then if we require to copy the state of one object to another object at the point of creation itself, copy constructors serve the purpose. Copy constructors are thus used to copy one object's state into another object at the point of creation or instantiation. For example, the statement

```
IntArray o1 (8);
cin >> a;
IntArray o2 (o1);
```

In the above statements, we create an `IntArray` object of size 8 and then read in values using the overloaded >> operator. The next statement creates another instance o2 of IntArray and copies the state of `o1` onto `o2` using the syntax `o2(o1)`. Thus, when the compiler sees the expression `o2(o1)` it invokes the copy constructor of the class to copy or assign the state of `o1 to o2`. Obviously, in this case, the object that is being copied (destination) is of the same size as that of the source object `(o1)`. Again copy constructors can either be user (programmer-defined) or default (compiler-defined). Well, with default copy constructors there is no guarantee that the destination object is in a consistent state. Copy constructors accept as argument the source object and this has to be only by reference. The reason shall be explained in a short while. The signature for copy constructor of class `IntArray` is as follows:

```
IntArray (const IntArray &)
```

The definition of the above copy constructor is as follows:

```
IntArray (const IntArray & source)
{
size = source.size;
aptr = new int [size];
```

```
for (int i=0;i<size;i++)
aptr [i]= source.aptr[i];
}
```

The copy constructor definition is quiet similar to the = operator overload. First, we copy the size of the source object onto the destination object size data member. Next, the memory is allocated for the required size of array elements. Then, we copy the elements of the source object array onto the destination object array using a `for loop` as shown above. Well, the definition of the copy constructor is quiet similar to the = operator definition, however, there is a difference. Now we shall explain the reasoning behind copy constructors accepting arguments only by reference.

Copy constructors can be passed arguments only by reference and pass by value is not allowed. There is a subtle reasoning behind it. If arguments were allowed to be passed by value to copy constructors, then as per the pass by value mechanism a local or duplicate copy of the arguments should be created and it is over this copy that local updations are performed. But then there is a problem with copy constructors. Copy constructors (function) accept as argument the source object. Now, if it is to be a by value syntax, then a local copy of the source object should be created. The very purpose of copy constructors is to copy the state of the source object on the destination object. Now, if copy constructors were to accept arguments by value, then for a local or temporary copy of the object to be created, the copy constructor for the argument that is an object should be called. This is nothing but the copy constructor function getting called within itself (direct recursion) and this recursion is infinite or based on no condition. Thus, if copy constructors are allowed to accept arguments by value, then it leads to indefinite recursion. Hence, the reasoning behind it is not allowing the copy constructors to accept arguments by reference.

Another point to be noted with copy constructors is that the copy constructor should not simply copy the source object pointer onto the destination object pointer [address]. This may solve our purpose of copying the source object contents into destination object, since both the source as well as the destination object variable would just be an alias for the source object's address. This might simulate the copying operation, but this can lead to serious consequences. The consequence is that assuming that the destination object is the most recently constructed object in comparison with the source object, the destructor of the destination object that would reclaim the objects memory allocation in this case would delete the storage space associated with both the target and the source object. Thus, further references to the memory location via the source object will be undefined. This serious consequence is also referred to as dangling pointer situation n and can cause serious run time errors.

Note: If both the copy constructor and the overloaded assignment operator are provided in a class then based on the use either the copy constructor or the overloaded assignment operator is called. Thus in case where an object's content is to be copied to another object at the point of object creation using one of the following syntaxes mentioned below calls the copy constructor of the class. Copy constructor application syntax is as follows:

```
IntArray o2(o1) ; or Array o2 = o1;
```

After having declared objects of the class, then trying to copy or assign the state of one object to the other using an = operator leads a call to the overloaded assignment operator of the class. Objects can be prevented from being copied or assigned to one another. All

we need to do is to make the copy constructor or the overloaded assignment operator of the respective class private members. Doing so prevents such member functions being invoked from outside the class, even via objects of the class. Private members are not accessible from outside the class, even via objects of the class.The overall C++ code that implements the array class overloading, including the operator overload functions and user instantiations is shown in Figure 6.2.

```
"intarray.h"
class IntArray
{
friend ostream & operator << (ostream &, const IntArray &);
friend istream & operator >> (istream &, IntArray &);
public:
IntArray(int=20);
IntArray(const IntArray &);
~IntArray();
const IntArray & operator =(const IntArray &);
bool operator ==(const IntArray &)const;
bool operator '=(const IntArray &)const;
int & operator[](int);
const int & operator [](int) const;
private:
int *aptr;
int size;
};
"intarray.cpp"
#include"intarray.h"
#include<iostream.h>
IntArray::IntArray(int sz)
{
size = sz;
aptr = new int [size];
++ arraycount;
for (int i=0;i< size; i++)
aptr[i]=0;
}
IntArray::~IntArray()
{
delete [ ] aptr;
};
IntArray::IntArray (const IntArray & source)
{
size = source.size;
aptr = new int [size];
for (int i=0;i<size;i++)
aptr [i]= source.aptr[i];
}
const IntArray & IntArray::operator = (const IntArray & rt)
{
```

Figure 6.2: Continued

```
if (&right != this)
{
if(size !=rt.size)
{
delete [] aptr;
size = rt.size;
aptr = new int [size];
}
for (int i=0;i<size;i++)
aptr [i]= rt.aptr[i];
}
else
return * this;
}
bool IntArray::operator == (const IntArray & r) const
{
if (size ! = r.size)
return FALSE;
for (int i=0;i<size;i++)
if (aptr[i] !=r.aptr[i])
return FALSE;
return TRUE;
}
bool IntArray:operator != (const IntArray &) const
{
return ! ( *this == right);
}
ostream & operator << (ostream & o,const IntArray & a)
{
for (int i=0;i<a.size;i++)
o << a.aptr[i];
return o; }
istream & operator >> (istream & i,IntArray & a)
{
for (int i=0;i<a.size;i++)
i >> a.aptr[i];
return i; }
int & operator [ ] (int sscript)
{
if (sscript >=0 && sscript < size)
return aptr[sscript];
else
abort();
}
const int operator [ ] (int sscript) const
{
if (sscript >= 0 && sscript < size)
return aptr[sscript];
else
abort();
}
```

Figure 6.2: Continued

```
"main.cpp"
#include<iostream.h>
#include"intarray.h"
int main()
{
IntArray a1(6),a2,a3(6),a4(5);
//check the intial states of the array objects
cout<<a1<<"\n"<<a2<<"\n"<<a3<<"\n"<<a4<<"\n";
//read input values for the array objects
cin>>a1>>a2>>>a3>>a4;
//assign one array object to another
a3=a1;
cout<<a3;
//check if two arrays are equal or not
if (a1==a3)
cout<<"\n Array Objects are Equal"<<endl;
if(a1!=a4)
cout<<"\n Array Objects are Not Equal"<<endl;
//copy constructor to create copy a5 from a1
IntArray a5(a1);
//Array Indexing- can try out for out of bounds as well
cout<<"\n Array Indexing"<<a1[3]<<"\n;
return 0;
}
```

Figure 6.2: C++ code for `IntArray` class overloading.

6.7 Conversion Constructors

The next types of constructors are the conversion constructors. Conversion is an inherent requirement while programming. Even with structured programming languages, casting operations were supported to allow conversion of one data type to the other. At any or all stages of programming, we would require type conversion. With object-oriented languages, the conversion operation would involve objects and might involve conversion between user-defined and built-in data types. Such conversion need not be defined by the programmer, if the conversion operation involves only built-in data types.

This is because the compiler is very well aware of how to perform conversions among built-in types. In fact, conversion involving built-in types are often referred to as casting. This lenience is available only when conversion involves built-in types. With user-defined object conversion, the programmer must specify the conversion behaviour. Functions that perform user-specified conversions are referred to as conversion constructors. In fact to be specific, single argument constructors that convert objects [user-defined or built-in type] into objects of a particular class are specifically referred to as conversion constructors. The vice versa operation of converting an object of one class to object of other class or built-in type is referred to as conversion operator or the cast operator function. The signature of the cast operator function for creating a temporary int object from an user-defined object of class `X` is as follows:

```
X:: operator int () const;
```

This syntax converts a user-defined object of class x into built type. The signature to convert user-defined object of class x into another user-defined object of class y is as follows:

```
X:: operator y() const;
```

Well, they are made as constant functions since the conversion is only temporary operation. Thus, it is treated only as a read operation. If o1 is object of type X and the compiler encounters the statement (int) o1, the compiler generates the call o1.operator int (); [based on signature: X: operator int () const]. Thus conversion constructors or cast operators serve the purpose of converting user-defined to predefined or user-defined conversions.

6.8 Overloading of Unary ++/-- Operators

We have seen how to overload the +, = and other operators. In this list of over loadable operators, ++ and - are also present. That is the unary increment and decrement operators can be overloaded. Now, ++/-- can be used either with the pre or postversion. Yes, the issue involved with overloading of increment and decrement operators is in differentiating whether it is used with pre or postmeanings. Let us assume that o is an instance of a class that has 3 integer data members. Let us assume that o is incremented. Now, the compiler will parse o++ and ++o as o.operator ++(), assuming that ++ is overloaded as a member function of the class.

Now, the compiler will be in a fix to differentiate the two member function calls, since their signatures are the same and the compiler will not be able to resolve the member function call or the compiler simply will not know which among the two functions to call, since both the functions have the same name operator ++; same signature in other words. Yes, this is precisely the issue of differentiating function signatures involved with the overloading of ++ and - operator. Well this signature differentiation issue is encountered even when ++ or -- is overloaded as a non-member function. O++ or ++o, both lead to the call or are parsed by the compiler as operator ++ (o).

Thus, to differentiate their signatures, the postincrement version is passed a dummy integer. In the case of member function overload of ++, the statement ++o is parsed as o. operator++(), while the statement o++ is parsed as o. operator ++(0), where 0 is the dummy integer to differentiate the two function calls. In the case of non-member function overload the compiler parses o++ or ++o as operator ++(o,0) and operator ++(o). The 0 argument is used by the compiler to differentiate the signatures of the preincrement and postincrement versions. The dummy 0 need not be passed by the programmer; it is implicitly passed by the compiler.

The programmer just needs to specify the dummy argument type (int) in the member function or non-member function signatures and function header of the respective function definitions. The code for unary operator overloading is shown in Figure 6.3. The pre and postincrement operator functions make use of a helper or utility function to increment the values of the data members by 1. Note how the preincrement operator function first calls the helper function to increment the object state by 1 and then returns the object's address (via the this pointer). Thus, first the increment is performed and then future assignments are handled. In the case of postincrement operator function, the current state of the object is copied to a temporary object and then the increment on the data members is done, resulting in first the assignment operation (old state) and then the increment being performed. State

of object of class A can be reflected by overloading the output operator to redirect the various data members onto the screen to appreciate the effect of pre and postincrement operators, more so when done as a part of an assignment operation.

```
1. class A {
2. private :
3. increment();
4. int no1;
5. int no2;
6. int no3;
7. public:
8. A (int =0,int=0,int=0);
9. ~A();
10. A & operator ++();
11. A & operator ++(int);};
12. A (int n1,int n2,int n3){
13. no1=n1;
14. no2=n2;
15. no3=n3;}
16. A & operator ++(){
17. increment ();
18. return * this; }
19. A & operator ++(int){
20. A temp = * this;
21. increment ();
22. return temp; }
23. void increment (){
24. no1 = no1 + 1;
25. no2 = no2 + 1;
26. no3 = no3 + 1;}
27. int main () {
28. A o1(5,6,7);
29. o1++;
30. ++o1;
31. return 0;}
```

Figure 6.3: Unary operator overloading example.

Review Questions

1. Justify the need for operator overloading in C++.
2. List a few built-in operators in C++ that carry overloaded meaning.
3. List the do's and don'ts of operator overloading.
4. How do you decide whether an operator is to be overloaded as a member or non-member function?
5. Appreciate the use of friend functions in the context of operator overloading.
6. What do you mean by arity of an operator?

7. For the Time class package created in earlier chapter (exercise) overload the input and output operators to read and display time object values.
8. Develop a C++ Matrix package and overload operators of + and - to perform matrix addition, subtraction operations respectively.
9. Develop a C++ class called Complex and simulate complex number arithmetic of addition and subtraction using operator overloading.
10. Why should copy constructor of a class accept arguments only by reference? Justify.
11. Differentiate copy from conversion constructor in C++ using an example.
12. What is the issue to be addressed while overloading unary operators?
13. For the Matrix package created earlier, overload the pre- and post-increment and decrement operators respectively.
14. Discuss the overloaded meanings of << and >> operators in C++.
15. Appreciate the `cout` and `cin` statement by interpreting the way the compiler parses `cout << a` and `cin >> a`.