UNIT – III

CLASSES AND OBJECTS – II

Topics covered

Classes & Objects –II: Friend functions, Passing objects as arguments, Returning objects, Arrays of objects, Dynamic objects, Pointers to objects, Copy constructors, Generic functions and classes, Applications Operator overloading using friend functions such as +, - , pre-increment, post-increment, [ ] etc., overloading <<, >>.

UNIT-3 Summary

In Part One, pointers and arrays were examined as they relate to C++'s built-in types. Here, they are discussed relative to objects. This chapter also looks at a feature related to the pointer called a *reference*. The chapter concludes with an examination of C++'s dynamic allocation operators.

1. Friend functions

It is possible to grant a nonmember function access to the private members of a class by using a **friend**. A **friend** function has access to all **private** and **protected** members of the class for which it is a **friend**. To declare a **friend** function, include its prototype within the class, preceding it with the keyword **friend**. Consider this program:

```
#include <iostream>
using namespace std;
class myclass {
int a, b;
public:
friend int sum(myclass x);
void set_ab(int i, int j);
  };
  void myclass::set_ab(int i, int j)
  {
  a = i;
  b = j;
  }
```

```
// Note: sum() is not a member function of any class.
int sum(myclass x)
{
/* Because sum() is a friend of myclass, it can
directly access a and b. */
return x.a + x.b;
}

int main()
{
myclass n;

n.set_ab(3, 4);
cout << sum(n);
return 0;
}
```

In this example, the **sum( )** function is not a member of **myclass**. However, it still has full access to its private members. Also, notice that **sum( )** is called without the use of the dot operator. Because it is not a member function, it does not need to be (indeed, it may not be) qualified with an object's name.

2. Passing objects as arguments

Objects may be passed to functions in just the same way that any other type of variable can. Objects are passed to functions through the use of the standard call-by-value mechanism. Although the passing of objects is straightforward, some rather unexpected events occur that relate to constructors and destructors. To understand why, consider this short program.

```
// Passing an object to a function.
<iostream>
namespace std;
class  myclass {
i;
public:
myclass(int n);
~myclass();
void set_i(int n) { i=n; }
int get_i() { return i; }
};
myclass::myclass(int n)
```

```
{
i = n;
cout << "Constructing " << i << "\n";
}
myclass::~myclass()
{
cout << "Destroying " << i << "\n";
}
void f(myclass ob);

main()
myclass

f(o);
cout <<
cout <<o.get_i() << "\n";

return 0;
}

void f(myclass ob)
{
ob.set_i(2);

cout << "This is local i: " << ob.get_i();
cout << "\n";
}
```

This program produces this output:

```
Constructing 1
This is local i: 2
Destroying 2
This is i in main: 1
Destroying 1
```

As the output shows, there is one call to the constructor, which occurs when **o** is created in **main( )**, but there are *two* calls to the destructor. Let's see why this is the case. When an object is passed to a function, a copy of that object is made (and this copy becomes the parameter in the function). This means that a new object comes into existence. When the function terminates, the copy of the argument (i.e., the parameter) is destroyed. This raises two fundamental questions: First, is the object's constructor called when the copy is made? Second, is the object's destructor called when the copy is destroyed? The answers may, at first, surprise you.

When a copy of an argument is made during a function call, the normal constructor is *not* called. Instead, the object's *copy constructor* is called. A copy constructor defines how a copy of an object is made. you can explicitly define a copy constructor for a class that you create . However, if a class does not explicitly define a copy constructor, as is the case here, then C++ provides one by default. The default copy constructor creates a bitwise (that is, identical) copy of the object. The reason a bitwise copy is made is easy to understand if you think about it. Since a normal constructor is used to initialize some aspect of an object, it must not be called to make a copy of an already existing object. Such a call would alter the contents of the object.

When passing an object to a function, you want to use the current state of the object, not its initial state. However, when the function terminates and the copy of the object used as an argument is destroyed, the destructor *is* called. This is necessary because the object has gone out of scope. This is why the preceding program had two calls to the destructor. The first was when the parameter to **f( )** went out-of-scope. The second is when **o** inside **main( )** was destroyed when the program ended.

To summarize: When a copy of an object is created to be used as an argument to a function, the normal constructor is not called. Instead, the default copy constructor makes a bit-by-bit identical copy. However, when the copy is destroyed (usually by going out of scope when the function returns), the destructor is called. Because the default copy constructor creates an exact duplicate of the original, it can, at times, be a source of trouble. Even though objects are passed to functions by means of the normal call-by-value parameter passing mechanism which, in theory, protects and insulates the calling argument, it is still possible for a side effect to occur that may affect, or even damage, the object used as an argument. For example, if an object used as an argument allocates memory and frees that memory when it is destroyed, then its local copy inside the function will free the same memory when its destructor is called. This will leave the original object damaged and effectively useless.

3. Returning objects

A function may return an object to the caller. For example, this is a valid C++ program:

```
// Returning objects from a function.
#include <iostream>
using namespace std;
class myclass {
int i;
public:
void set_i(int n) { i=n; }
int get_i() { return i; }
};
myclass f();   // return object of type myclass
int main()
{
myclass o;
o = f();
cout << o.get_i() << "\n";
return 0;
}
myclass f()
{
myclass x;
x.set_i(1);
return x;
}
```

When an object is returned by a function, a temporary object is automatically created that holds the return value. It is this object that is actually returned by the function. After the value has been returned, this object is destroyed. The destruction
of this temporary object may cause unexpected side effects in some situations. For example, if the object returned by the function has a destructor that frees dynamically allocated memory, that memory will be freed even though the object that is receiving the return value is still using it.

4. Arrays of objects

In C++, it is possible to have arrays of objects. The syntax for declaring and using an object array is exactly the same as it is for any other type of array. For example, this program uses a three-element array of objects:

```
#include <iostream>
using namespace std;
class cl {
int i;
public:
void set_i(int j) { i=j; }
int get_i() { return i; }
};
int main()
{
cl ob[3];
int i;
for(i=0; i<3; i++) ob[i].set_i(i+1);
for(i=0; i<3; i++)
cout << ob[i].get_i() << "\n";
return 0;
}
```

This program displays the numbers **1**, **2**, and **3** on the screen. If a class defines a parameterized constructor, you may initialize each object in an array by specifying an initialization list, just like you do for other types of arrays. However, the exact form of the initialization list will be decided by the number of parameters required by the object's constructors. For objects whose constructors have only one parameter, you can simply specify a list of initial values, using the normal array-initialization syntax. As each element in the array is created, a value from the list is passed to the constructor's parameter. For example, here is a slightly different version of the preceding program that uses an initialization:

```
#include <iostream>
using namespace std;
class cl {
int i;
public:
cl(int j) { i=j; } // constructor
int get_i() { return i; }
};
int main()
{
cl ob[3] = {1, 2, 3}; // initializers
int i;
for(i=0; i<3; i++)
cout << ob[i].get_i() << "\n";
return 0;
}
```

As before, this program displays the numbers **1**, **2**, and **3** on the screen. Actually, the initialization syntax shown in the preceding program is shorthand for this longer form:

```
cl ob[3] = { cl(1), cl(2), cl(3) };
```

Here, the constructor for **cl** is invoked explicitly. Of course, the short form used in the program is more common. The short form works because of the automatic conversion that applies to constructors taking only one argument. Thus, the short form can only be used to initialize object arrays whose constructors only require one argument. If an object's constructor requires two or more arguments, you will have to use the longer initialization form. For example,

```
#include <iostream>
using namespace std;
class cl {
int h;
int i;
public:
cl(int j, int k) { h=j; i=k; } // constructor with 2 parameters
int get_i() {return i;}
int get_h() {return h;}
};
int main()
{
cl ob[3] = {
cl(1, 2), // initialize
cl(3, 4),
cl(5, 6)
};

int i;
for(i=0; i<3; i++) {
cout << ob[i].get_h();
cout << ", ";
cout << ob[i].get_i() << "\n";
}
return 0;
}
```

Here, **cl**'s constructor has two parameters and, therefore, requires two arguments. This means that the shorthand initialization format cannot be used and the long form, shown in the example, must be employed.

5. Dynamic objects

C++ allows you to generate a special type of pointer that "points" generically to a member of a class, not to a specific instance of that member in an object. This sort of pointer is called a *pointer to a class member* or a *pointer-to-member*, for short. A pointer to a member is not the same as a normal C++ pointer. Instead, a pointer to a member provides only an offset into an object of the member's class at which that member can be found. Since member pointers are not true pointers, the **.** and –> cannot be applied to them. To access a member of a class given a pointer to it, you must use the special pointer-to-member operators **.*** and **–>***. Their job is to allow you to access a member of a class given a pointer to that member.

C++ provides two dynamic allocation operators: **new** and **delete**. These operators are

used to allocate and free memory at run time. Dynamic allocation is an important of almost all real-world programs. As explained in Part One, C++ also supports dynamic memory allocation functions, called **malloc( )** and **free( ).** These are included for the sake of compatibility with C. However, for C++ code, you should use the **new** and **delete** operators because they have several advantages.

The **new** operator allocates memory and returns a pointer to the start of it. The **delete** operator frees memory previously allocated using **new**. The general forms of **new** and **delete** are shown here:

p_var = new *type*;
delete *p_var*;

Here, *p_var* is a pointer variable that receives a pointer to memory that is large enough to hold an item of type *type*.

Since the heap is finite, it can become exhausted. If there is insufficient available memory to fill an allocation request, then **new** will fail and a **bad_alloc** exception will be generated. This exception is defined in the header **<new>**. Your program should handle this exception and take appropriate action if a failure occurs. If this exception is not handled by your program, then your program will be terminated. The actions of **new** on failure as just described are specified by Standard C++. The trouble is that not all compilers, especially older ones, will have implemented **new** in compliance with Standard C++. When C++ was first invented, **new** returned null on failure. Later, this was changed such that **new** caused an exception on failure. Finally, it was decided that a **new** failure will generate an exception by default, but that a null pointer could be returned instead, as an option. Thus, **new** has been implemented differently, at different times, by compiler manufacturers. Although all compilers will eventually implement **new** in compliance with Standard C++, currently the only way to know the precise action of **new** on failure is to check your compiler's documentation.

Since Standard C++ specifies that **new** generates an exception on failure, this is the way the code in this book is written. If your compiler handles an allocation failure differently, you will need to make the appropriate changes. Here is a program that allocates memory to hold an integer:

```
#include <iostream>
#include <new>
using namespace std;
int main()
{
int *p;
try {
p = new int; // allocate space for an int
(bad_alloc xa) {
<< "Allocation Failure\n";
return 1;
}
              } catch
                cout
*p = 100;
cout << "At " << p << " ";
cout << "is the value " << *p << "\n";
delete p;
return 0;
}
```

This program assigns to **p** an address in the heap that is large enough to hold an integer. It then assigns that memory the value 100 and displays the contents of the memory on the screen. Finally, it frees the dynamically allocated memory. Remember, if your compiler implements **new** such that it returns null on failure, you must change the preceding program appropriately. The **delete** operator must be used only with a valid pointer previously allocated by using **new**. Using any other type of pointer with **delete** is undefined and will almost certainly cause serious problems, such as a system crash. Although **new** and **delete** perform functions similar to **malloc( )** and **free( )**, they have several advantages. First, **new** automatically allocates enough memory to hold an object of the specified type. You do not need to use the **sizeof** operator. Because the size is computed automatically, it eliminates any possibility for error in this regard. Second, **new** automatically returns a pointer of the specified type. You don't need to use an explicit type cast as you do when allocating memory by using **malloc( )**. Finally, both **new** and **delete** can be overloaded, allowing you to create customized allocation systems.

Although there is no formal rule that states this, it is best not to mix **new** and **delete** with **malloc( )** and **free( )** in the same program. There is no guarantee that they are mutually compatible.

6. Pointers to objects

Just as you can have pointers to other types of variables, you can have pointers to objects. When accessing members of a class given a pointer to an object, use the arrow (–>) operator instead of the dot operator. The next program illustrates how to access an object given a pointer to it:

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

class cl {
int i;
public:
cl(int j) { i=j; }
int get_i() { return i; }
};
int main()
{
cl ob(88), *p;
p = &ob; // get address of ob
cout << p->get_i(); // use -> to call get_i()
return 0;
}
```

As you know, when a pointer is incremented, it points to the next element of its type. For example, an integer pointer will point to the next integer. In general, all pointer arithmetic is relative to the base type of the pointer. (That is, it is relative to the type of data that the pointer is declared as pointing to.) The same is true of pointers to objects. For example, this program uses a pointer to access all three elements of array **ob** after being assigned **ob**'s starting address:

```
#include <iostream>
using namespace std;
class cl {
int i;
public:
```

```
cl() { i=0; }
cl(int j) { i=j; }
int get_i() { return i; }
};
int main()
{
cl ob[3] = {1, 2, 3};
cl *p;
int i;
p = ob; // get start of array
for(i=0; i<3; i++) {
cout << p->get_i() << "\n";
p++; // point to next object
}
return 0;
}
```

You can assign the address of a public member of an object to a pointer and then access that member by using the pointer. For example, this is a valid C++ program that displays the number **1** on the screen:

```
#include <iostream>
using namespace std;
class cl {
public:
int i;
cl(int j) { i=j; }
};
int main()
{
cl ob(1);
int *p;
p = &ob.i; // get address of ob.i
cout << *p; // access ob.i via p
return 0;
}
```

Because **p** is pointing to an integer, it is declared as an integer pointer. It is irrelevant that **i** is a member of **ob** in this situation.

7. Copy constructors

One of the more important forms of an overloaded constructor is the *copy constructor*. Defining a copy constructor can help you prevent problems that might occur when one object is used to initialize another. Let's begin by restating the problem that the copy constructor is designed to solve. By default, when one object is used to initialize another, C++ performs a bitwise copy. That is, an identical copy of the initializing object is created in the target object.

Although this is perfectly adequate for many cases—and generally exactly what you want to happen—there are situations in which a bitwise copy should not be used. One of the most common is when an object allocates memory when it is created. For example, assume a class called *MyClass* that allocates memory for each object when it is created, and an object *A* of that class. This means that *A* has already allocated its memory. Further, assume that *A* is used to initialize *B*, as shown here:

```
MyClass B = A;
```

If a bitwise copy is performed, then *B* will be an exact copy of *A*. This means that *B* will be using the same piece of allocated memory that *A* is using, instead of allocating its own. Clearly, this is not the desired outcome. For example, if *MyClass* includes a destructor that frees the memory, then the same piece of memory will be freed twice when *A* and *B* are destroyed!

The same type of problem can occur in two additional ways: first, when a copy of an object is made when it is passed as an argument to a function; second, when a temporary object is created as a return value from a function. Remember, temporary objects are automatically created to hold the return value of a function and they may also be created in certain other circumstances.

To solve the type of problem just described, C++ allows you to create a copy constructor, which the compiler uses when one object initializes another. Thus, your copy constructor bypasses the default bitwise copy. The most common general form of a copy constructor is

```
classname (const classname &o) {
// body of constructor
}
```

Here, *o* is a reference to the object on the right side of the initialization. It is permissible for a copy constructor to have additional parameters as long as they have default arguments defined for them. However, in all cases the first parameter must be a reference to the object doing the initializing.

It is important to understand that C++ defines two distinct types of situations in which the value of one object is given to another. The first is assignment. The second is initialization, which can occur any of three ways:

- When one object explicitly initializes another, such as in a declaration
- When a copy of an object is made to be passed to a function
- When a temporary object is generated (most commonly, as a return

value)

The copy constructor applies only to initializations. For example, assuming a class called **myclass**, and that **y** is an object of type **myclass**, each of the following statements involves initialization.

```
myclass x = y; // y explicitly initializing x
func(y);       // y passed as a parameter
y = func();    // y receiving a temporary, return object
```

Following is an example where an explicit copy constructor is needed. This program creates a very limited "safe" integer array type that prevents array boundaries from being overrun. Storage for each array is allocated by the use of **new**, and a pointer to the memory is maintained within each array object.

```
/* This program creates a "safe" array class. Since space
for the array is allocated using new, a copy constructor
is provided to allocate memory when one array object is
used to initialize another.
*/
#include <iostream>
#include <new>
#include <cstdlib>
using namespace std;

class array {
int *p;
int size;
public:
array(int sz) {
try {
p = new int[sz];
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
exit(EXIT_FAILURE);
}
size = sz;
}
~array() { delete [] p; }

// copy constructor
```

```
array(const array &a);

void put(int i, int j) {
if(i>=0 && i<size) p[i] = j;
}
int get(int i) {
return p[i];
}
};
// Copy Constructor
array::array(const array &a) {
int i;
try {
p = new int[a.size];
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
exit(EXIT_FAILURE);
}
for(i=0; i<a.size; i++) p[i] = a.p[i];
}
int main()
{
array num(10);
int i;
for(i=0; i<10; i++)  num.put(i,
for(i=9; i>=0; i--)  cout << num.get(i);
cout << "\n";
// create another array and initialize with num
array x(num); // invokes copy constructor
for(i=0; i<10; i++) cout << x.get(i);

return 0;
}
```

Let's look closely at what happens when **num** is used to initialize **x** in the statement

array x(num); // invokes copy constructor

The copy constructor is called, memory for the new array is allocated and stored in **x.p**, and the contents of **num** are copied to **x**'s array. In this way, **x** and **num** have arrays that contain the same values, but each array is separate and distinct. (That is, **num.p** and **x.p** do not point to the same piece of memory.) If the copy constructor had not been created, the default bitwise initialization would have resulted in **x** and **num** sharing the same memory for their arrays. (That is, **num.p** and **x.p** would have indeed pointed to the same location.)

8. Generic functions and classes

C++ allows a function to assign a parameter a default value when no argument corresponding to that parameter is specified in a call to that function. The default value is specified in a manner syntactically similar to a variable initialization. For example, this declares **myfunc( )** as taking one **double** argument with a default value of 0.0:

```
void myfunc(double d = 0.0)
{
// ...
}
```

Now, **myfunc( )** can be called one of two ways, as the following examples show:

```
myfunc(198.234); // pass an explicit value
myfunc();          // let function use default
```

The first call passes the value 198.234 to **d**. The second call automatically gives **d** the default value zero. One reason that default arguments are included in C++ is because they provide another method for the programmer to manage greater complexity. To handle the widest variety of situations, quite frequently a function contains more parameters than are required for its most common usage. Thus, when the default arguments apply, you need specify only the arguments that are meaningful to the exact situation, not all those needed by the most general case. For example, many of the C++ I/O functions make use of default arguments for just this reason.

A simple illustration of how useful a default function argument can be is shown by the **clrscr( )** function in the following program. The **clrscr( )** function clears the screen by outputting a series of linefeeds (not the most efficient way, but sufficient for this example). Because a very common video mode displays 25 lines of text, the default argument of 25 is provided. However, because some video modes display more or less than 25 lines, you can override the default argument by specifying one explicitly.

9. Applications Operator overloading using friend functions such as +, - , pre-increment, post-increment, [ ] etc., overloading <<, >>.

A member operator function takes this general form:

```
ret-type class-name::operator#(arg-list)
{
// operations
}
```

Often, operator functions return an object of the class they operate on, but *ret-type* can be any valid type. The # is a placeholder. When you create an operator function, substitute the operator for the #. For example, if you are overloading the / operator, use **operator/.** When you are overloading a unary operator, *arg-list* will be empty. When you are overloading binary operators, *arg-list* will contain one parameter.

(The reasons for this seemingly unusual situation will be made clear in a moment.)

Here is a simple first example of operator overloading. This program creates a class called **loc**, which stores longitude and latitude values. It overloads the + operator relative to this class. Examine this program carefully, paying special attention to the definition of **operator+( )**:

```
#include<iostream>
namespace std;
Class loc {
longitude, latitude;
public:
loc() {}
loc(int lg, int lt) {
longitude = lg;
latitude    lt;
}
void show() {
cout << longitude   << " ";
cout << latitude    << "\n";
}
loc operator+(loc op2);
};
// Overload + for loc.
loc loc::operator+(loc op2)
{
loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp;
}
```

```
int main()
{
loc ob1(10, 20), ob2( 5, 30);
ob1.show(); // displays 10 20
ob2.show(); // displays 5 30

ob1 = ob1 + ob2;
ob1.show(); // displays 15 50
return 0;
}
```

As you can see, **operator+( )** has only one parameter even though it overloads the binary + operator. (You might expect two parameters corresponding to the two operands of a binary operator.) The reason that **operator+( )** takes only one parameter is that the operand on the left side of the + is passed implicitly to the function through the **this** pointer. The operand on the right is passed in the parameter **op2**. The fact that the left operand is passed using **this** also implies one important point: When binary operators are overloaded, it is the object on the left that generates the call to the operator function. As mentioned, it is common for an overloaded operator function to return an object of the class it operates upon. By doing so, it allows the operator to be used in larger expressions. For example, if the **operator+( )** function returned some other type, this expression would not have been valid:

```
ob1 = ob1 + ob2;
```

In order for the sum of **ob1** and **ob2** to be assigned to **ob1**, the outcome of that operation must be an object of type **loc**. Further, having **operator+( )** return an object of type **loc** makes possible the following statement:

```
(ob1+ob2).show(); // displays outcome of ob1+ob2
```

In this situation, **ob1+ob2** generates a temporary object that ceases to exist after the call to **show( )** terminates.

It is important to understand that an operator function can return any type and that the type returned depends solely upon your specific application. It is just that, often, an operator function will return an object of the class upon which it operates.

One last point about the **operator+( )** function: It does not modify either operand. Because the traditional use of the + operator does not modify either operand, it makes sense for the overloaded version not to do so either. (For example, 5+7 yields 12, but neither 5 nor 7 is changed.) Although you are free to perform any operation you want inside an operator function, it is usually best to stay within the context of the normal use of the operator.

The next program adds three additional overloaded operators to the **loc** class: the –, the =, and the unary ++. Pay special attention to how these functions are defined.

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {} // needed to construct temporaries
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator+(loc op2);
loc operator-(loc op2);
loc operator=(loc op2);
loc operator++();
};
// Overload + for loc.
loc loc::operator+(loc op2)
{
loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp;
}
// Overload - for loc.
loc loc::operator-(loc op2)
{
loc temp;
// notice order of operands
temp.longitude = longitude - op2.longitude;
temp.latitude = latitude - op2.latitude;
return temp;
}
```

```
// Overload asignment for loc.loc loc::operator=(loc op2)
{
longitude = op2.longitude;
latitude = op2.latitude;

return *this; // i.e., return object that generated call
}

// Overload prefix ++ for loc.
loc loc::operator++()
{
longitude++;
latitude++;
return *this;
}
int main()
{
loc ob1(10, 20), ob2( 5, 30), ob3(90, 90);
ob1.show();
ob2.show();
++ob1;
ob1.show(); // displays 11 21
ob2 = ++ob1;
ob1.show(); // displays 12 22
ob2.show(); // displays 12 22
ob1 = ob2 = ob3; // multiple assignment
ob1.show(); // displays 90 90
ob2.show(); // displays 90 90

return 0;
}
```

First, examine the **operator–( )** function. Notice the order of the operands in the subtraction. In keeping with the meaning of subtraction, the operand on the right side of the minus sign is subtracted from the operand on the left. Because it is the object on the left that generates the call to the **operator–( )** function, **op2**'s data must be subtracted from the data pointed to by **this**. It is important to remember which operand generates the call to the function.

In C++, if the = is not overloaded, a default assignment operation is created automatically for any class you define. The default assignment is simply a member- by-member, bitwise copy. By overloading the =, you can define explicitly what the assignment does relative to a class. In this example, the overloaded = does exactly the same thing as the default, but in other situations, it could perform other operations. Notice that the **operator=( )** function returns ***this**, which is the object that generated the call. This arrangement is necessary if you want to be able to use multiple assignment operations such as this:

```
ob1 = ob2 = ob3; // multiple assignment
```

Now, look at the definition of **operator++( ).** As you can see, it takes no parameters. Since ++ is a unary operator, its only operand is implicitly passed by using the **this** pointer.

Notice that both **operator=( )** and **operator++( )** alter the value of an operand. In the case of assignment, the operand on the left (the one generating the call to the **operator=( )** function) is assigned a new value. In the case of the ++, the operand is incremented. As stated previously, although you are free to make these operators do anything you please, it is almost always wisest to stay consistent with their original meanings.

In the preceding program, only the prefix form of the increment operator was overloaded. However, Standard C++ allows you to explicitly create separate prefix and postfix versions of the increment or decrement operators. To accomplish this, you must define two versions of the **operator++( )** function. One is defined as shown in the foregoing program. The other is declared like this:

```
loc operator++(int x);
```

If the ++ precedes its operand, the **operator++( )** function is called. If the ++ follows its operand, the **operator++(int x)** is called and **x** has the value zero. The preceding example can be generalized. Here are the general forms for the prefix and postfix ++ and – – operator functions.

```
// Prefix increment
type operator++( ) {
// body of prefix operator
}
// Postfix increment
type operator++(int x) {
// body of postfix operator
}
// Prefix decrement
type operator– –( ) {
// body of prefix operator
}
```

```
// Postfix decrement
type operator– –(int x) {
// body of postfix operator
}
```

9.1 Overloading [ ]

In C++, the **[ ]** is considered a binary operator when you are overloading it. Therefore, the general form of a member **operator[ ]( )** function is as shown here:

```
type class-name::operator[](int i)
{
// . . .
}
```

Technically, the parameter does not have to be of type **int**, but an **operator[ ]( )** function is typically used to provide array subscripting, and as such, an integer value is generally used

Given an object called **O**, the expression

```
O[3]
```

translates into this call to the **operator[ ]( )** function:

```
O.operator[](3)
```

That is, the value of the expression within the subscripting operators is passed to the **operator[ ]( )** function in its explicit parameter. The **this** pointer will point to **O**, the object that generated the call. In the following program, **atype** declares an array of three integers. Its constructor initializes each member of the array to the specified values. The overloaded **operator[ ]( )** function returns the value of the array as indexed by the value of its parameter.

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

class atype {
int a[3];
public:
atype(int i, int j, int k) {
a[0] = i;
```

```
            a[1] = j;
            a[2] = k;
        }
        int operator[](int i) { return a[i]; }
        };

int main()
{
atype ob(1, 2, 3);
cout << ob[1]; // displays 2
return 0;
}
```

You can design the **operator[ ]( )** function in such a way that the **[ ]** can be used on both the left and right sides of an assignment statement. To do this, simply specify the return value of **operator[ ]( )** as a reference. The following program makes this change and shows its use:

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

        class atype {
        int a[3];
        public:
        atype(int i, int j, int k) {
        a[0] = i;
        a[1] = j;
        a[2] = k;
        }
        int &operator[](int i) { return a[i]; }
        };
    int main()
    {
    atype ob(1, 2, 3);
    cout << ob[1]; // displays 2
    cout << " ";
    ob[1] = 25; // [] on left of =
    cout << ob[1]; // now displays 25
    return 0;
}
```

Because **operator[ ]( )** now returns a reference to the array element indexed by **i,** it can be used on the left side of an assignment to modify an element of the array. (Of course, it may still be used on the right side as well.)

One advantage of being able to overload the **[ ]** operator is that it allows a means of implementing safe array indexing in C++. As you know, in C++, it is possible to overrun (or underrun) an array boundary at run time without generating a run-time error message. However, if you create a class that contains the array, and allow access to that array only through the overloaded **[ ]** subscripting operator, then you can intercept an out-of-range index. For example, this program adds a range check to the preceding program and proves that it works:

```
// A safe array example.
#include <iostream>
#include <cstdlib>
using namespace std;
class atype {
int a[3];
public:
atype(int i, int j, int k) {
a[0] = i;
a[1] = j;
a[2] = k;
}
int &operator[](int i);
};
// Provide range checking for atype.
int &atype::operator[](int i)
{
if(i<0 || i> 2) {
cout << "Boundary Error\n";
exit(1);
}
return a[i];
}
int main()
{
atype ob(1, 2, 3);
cout << ob[1]; // displays 2
cout << " ";
ob[1] = 25; // [] appears on left
cout << ob[1]; // displays 25
ob[3] = 44; // generates runtime error, 3 out-of-range
return 0;
```

} In this program, when the statement

```
ob[3] = 44;
```

executes, the boundary error is intercepted by **operator[]( ),** and the program is terminated before any damage can be done. (In actual practice, some sort of error-handling function would be called to deal with the out-of-range condition; the program would not have to terminate.)

9.2 Overloading –>

The –> pointer operator, also called the *class member access* operator, is considered a unary operator when overloading. Its general usage is shown here:

object->element;

Here, *object* is the object that activates the call. The **operator–>( )** function must return

a pointer to an object of the class that **operator–>( )** operates upon. The *element* must be some member accessible within the object. The following program illustrates overloading the –> by showing the equivalence between **ob.i** and **ob–>i** when **operator–>( )** returns the **this** pointer:

```
#include <iostream>
using namespace std;
class myclass {
public:
int i;
myclass *operator->() {return this;}
};
int main()
{
myclass ob;
ob->i = 10; // same as ob.i
cout << ob.i << " " << ob->i;
return 0;
}
```

An **operator–>( )** function must be a member of the class upon which it works. The general skeleton of an overloaded output operator is

```
// general skeleton of the overloaded output operator
ostream&
```

```
operator <<(ostream& os, const ClassType &object)
{
// any special logic to prepare object
// actual output of members
os << // ...
// return ostream object
return os;
}
```

We can now write the Sales_item output operator:

```
ostream&
operator<<(ostream& out, const Sales_item& s)
{
out << s.isbn << "\t" << s.units_sold << "\t"
<< s.revenue << "\t" << s.avg_price();
return out;
}
```

The Sales_item Input Operator

The Sales_item input operator looks like:

```
istream&
operator>>(istream& in, Sales_item& s)
{
double price;
in >> s.isbn >> s.units_sold >> price;
// check that the inputs succeeded
if (in)
s.revenue = s.units_sold * price;
else
s = Sales_item(); // input failed: reset object to default state
return in;
}
```