Module 9

A Closer Look at Classes

Table of Contents

This module continues the discussion of the class begun in Module 8. It examines a number of class-related topics, including overloading constructors, passing objects to functions, and returning objects. It also describes a special type of constructor, called the copy constructor, which is used when a copy of an object is needed. Next, friend functions are described, followed by structures and unions, and the 'this' keyword. The module concludes with a discussion of operator overloading, one of C++'s most exciting features.

CRITICAL SKILL 9.1: Overloading Constructors

Although they perform a unique service, constructors are not much different from other types of functions, and they too can be overloaded. To overload a class' constructor, simply declare the various forms it will take. For example, the following program defines three constructors:

```
// Overload the constructor.

#include <iostream>
using namespace std;

class Sample {
public:
  int x;
  int y;

  // Overload the default constructor.
  Sample() { x = y = 0; }

  // Constructor with one parameter.
  Sample(int i) { x = y = i; }

  // Constructor with two parameters.
  Sample(int i, int j) { x = i; y = j; }
};

int main() {
  Sample t;          // invoke default constructor
  Sample t1(5);      // use Sample(int)
  Sample t2(9, 10); // use Sample(int, int)

  cout << "t.x: " << t.x << ", t.y: " << t.y << "\n";
  cout << "t1.x: " << t1.x << ", t1.y: " << t1.y << "\n";
  cout << "t2.x: " << t2.x << ", t2.y: " << t2.y << "\n";

  return 0;
}
```

Overload the **Sample** constructor.

The output is shown here:

```
t.x: 0, t.y: 0
t1.x: 5, t1.y: 5
t2.x: 9, t2.y: 10
```

This program creates three constructors. The first is a parameterless constructor, which initializes both x and y to zero. This constructor becomes the default constructor, replacing the default constructor supplied automatically by C++. The second takes one parameter, assigning its value to both x and y. The third constructor takes two parameters, initializing x and y individually.

Overloaded constructors are beneficial for several reasons. First, they add flexibility to the classes that you create, allowing an object to be constructed in a variety of ways. Second, they offer convenience to the user of your class by allowing an object to be constructed in the most natural way for the given task. Third, by defining both a default constructor and a parameterized constructor, you allow both initialized and uninitialized objects to be created.

CRITICAL SKILL 9.2: Assigning Objects

If both objects are of the same type (that is, both are objects of the same class), then one object can be assigned to another. It is not sufficient for the two classes to simply be physically similar—their type names must be the same. By default, when one object is assigned to another, a bitwise copy of the first object's data is assigned to the second. Thus, after the assignment, the two objects will be identical, but separate. The following program demonstrates object assignment:

```
// Demonstrate object assignment.

#include <iostream>
using namespace std;

class Test {
  int a, b;
public:
  void setab(int i, int j) { a = i, b = j; }
  void showab() {
    cout << "a is " << a << '\n';
    cout << "b is " << b << '\n';
  }
};

int main()
{
  Test ob1, ob2;

  ob1.setab(10, 20);
  ob2.setab(0, 0);
  cout << "ob1 before assignment: \n";
  ob1.showab();
  cout << "ob2 before assignment: \n";
  ob2.showab();
  cout << '\n';

  ob2 = ob1; // assign ob1 to ob2 <-------- Assign one object to another. //
```

```
  cout << "ob1 after assignment: \n";
  ob1.showab();
  cout << "ob2 after assignment: \n";
  ob2.showab();
  cout << '\n';

  ob1.setab(-1, -1); // change ob1

  cout << "ob1 after changing ob1: \n";
  ob1.showab();
  cout << "ob2 after changing ob1: \n";
  ob2.showab();

  return 0;
}
```

This program displays the following output:

```
ob1 before assignment:
a is 10
b is 20
ob2 before assignment:
a is 0
b is 0

ob1 after assignment:
a is 10
b is 20
ob2 after assignment:
a is 10
b is 20

ob1 after changing ob1:
a is -1
b is -1
ob2 after changing ob1:
a is 10
b is 20
```

As the program shows, the assignment of one object to another creates two objects that contain the same values. The two objects are otherwise still completely separate. Thus, a subsequent modification of one object's data has no effect on that of the other. However, you will need to watch for side effects, which may still occur. For example, if an object A contains a pointer to some other object B, then when a copy of A is made, the copy will also contain a field that points to B. Thus, changing B will affect both objects. In situations like this, you may need to bypass the default bitwise copy by defining a custom assignment operator for the class, as explained later in this module.

CRITICAL SKILL 9.3: Passing Objects to Functions

An object can be passed to a function in the same way as any other data type. Objects are passed to functions using the normal C++ call-by-value parameter-passing convention. This means that a copy of the object, not the actual object itself, is passed to the function. Therefore, changes made to the object inside the function do not affect the object used as the argument to the function. The following program illustrates this point:

```
// Pass an object to a function.

#include <iostream>
using namespace std;

class MyClass {
  int val;
public:
  MyClass(int i) {
    val = i;
  }

  int getval() { return val; }
  void setval(int i) { val = i; }
};

void display(MyClass ob) <-- display( ) takes a MyClass object as a parameter.
{
  cout << ob.getval() << '\n';
}

void change(MyClass ob) <-- change( ) also takes a MyClass object as a parameter.
{
  ob.setval(100); // no effect on argument

  cout << "Value of ob inside change(): ";
  display(ob);
}

int main()
{
  MyClass a(10);

  cout << "Value of a before calling change(): ";
  display(a); <-- Pass a MyClass object to display( ) and change( ).
  change(a);
  cout << "Value of a after calling change(): ";
  display(a);

  return 0;
}
```

The output is shown here:

```
Value of a before calling change(): 10
```

```
Value of ob inside change(): 100
Value of a after calling change(): 10
```
As the output shows, changing the value of ob inside change( ) has no effect on a inside main( ).

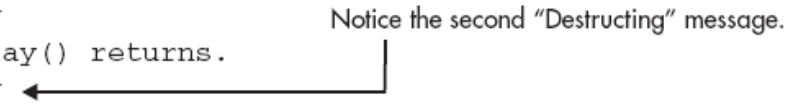
Constructors, Destructors, and Passing Objects

Although passing simple objects as arguments to functions is a straightforward procedure, some rather unexpected events occur that relate to constructors and destructors. To understand why, consider this short program:

```
// Constructors, destructors, and passing objects.

#include <iostream>
using namespace std;

class MyClass {
  int val;
public:
  MyClass(int i) {
    val = i;
    cout << "Inside constructor\n";
  }

  ~MyClass() { cout << "Destructing\n"; }
  int getval() { return val; }
};

void display(MyClass ob)
{
  cout << ob.getval() << '\n';
}

int main()
{
  MyClass a(10);

  cout << "Before calling display().\n";
  display(a);
  cout << "After display() returns.\n";

  return 0;
}
```

This program produces the following unexpected output:

```
Inside constructor
Before calling display().
10
Destructing
After display() returns.
Destructing
```



As you can see, there is one call to the constructor (which occurs when a is created), 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 (that is, 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. (Later in this module you will see how to create a copy constructor.)

However, if a class does not explicitly define a copy constructor, 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 function 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 display( ) went out of scope. The second is when a 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.

Passing Objects by Reference

Another way that you can pass an object to a function is by reference. In this case, a reference to the object is passed, and the function operates directly on the object used as an argument. Thus, changes made to the parameter will affect the argument, and passing an object by reference is not applicable to all situations. However, in the cases in which it is, two benefits result. First, because only an address to the object is being passed rather than the entire object, passing an object by reference can be much faster and more efficient than passing an object by value. Second, when an object is passed by

reference, no new object comes into existence, so no time is wasted constructing or destructing a temporary object.

Here is an example that illustrates passing an object by reference:

```
// Constructors, destructors, and passing objects.

#include <iostream>
using namespace std;

class MyClass {
  int val;
public:
  MyClass(int i) {
    val = i;
    cout << "Inside constructor\n";
  }

  ~MyClass() { cout << "Destructing\n"; }
  int getval() { return val; }
  void setval(int i) { val = i; }
};

void display(MyClass &ob)
{
  cout << ob.getval() << '\n';
}

void change(MyClass &ob)
{
  ob.setval(100);
}

int main()
{
  MyClass a(10);

  cout << "Before calling display().\n";
  display(a);
  cout << "After display() returns.\n";

  change(a);
  cout << "After calling change().\n";
  display(a);

  return 0;
}
```

Here, the **MyClass** object is passed by reference.

The output is

```
Inside constructor
Before calling display().
10
After display() returns.
After calling change().
100
Destructing
```

In this program, both display( ) and change( ) use reference parameters. Thus, the address of the argument, not a copy of the argument, is passed, and the functions operate directly on the argument. For example, when change( ) is called, a is passed by reference. Thus, changes made to the parameter ob in change( ) affect a in main( ). Also, notice that only one call to the constructor and one call to the destructor is made. This is because only one object, a, is created and destroyed. No temporary objects are needed by the program.

A Potential Problem When Passing Objects

Even when 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 allocates some system resource (such as memory) when it is created and frees that resource when it is destroyed, then its local copy inside the function will free that same resource when its destructor is called. This is a problem because the original object is still using this resource. This situation usually results in the original object being damaged.

One solution to this problem is to pass an object by reference, as shown in the preceding section. In this case, no copy of the object is made, and thus, no object is destroyed when the function returns. As explained, passing objects by reference can also speed up function calls, because only the address of the object is being passed. However, passing an object by reference may not be applicable to all cases. Fortunately, a more general solution is available: you can create your own version of the copy constructor. Doing so lets you define precisely how a copy of an object is made, allowing you to avoid the type of problems just described. However, before examining the copy constructor, let's look at another, related situation that can also benefit from a copy constructor.

CRITICAL SKILL 9.4: Returning Objects

Just as objects can be passed to functions, functions can return objects. To return an object, first declare the function as returning a class type. Second, return an object of that type using the normal return statement. The following program has a member function called mkBigger( ). It returns an object that gives val a value twice as large as the invoking object.

```
// Returning objects.

#include <iostream>
using namespace std;

class MyClass {
  int val;
```

```
public:
  // Normal constructor.
  MyClass(int i) {
    val = i;
    cout << "Inside constructor\n";
  }

  ~MyClass() {
     cout << "Destructing\n";
  }

  int getval() { return val; }

  // Return an object.
  MyClass mkBigger() {
    MyClass o(val * 2);

    return o;
  }
};

void display(MyClass ob)
{
  cout << ob.getval() << '\n';
}

int main()
{
  cout << "Before constructing a.\n";
  MyClass a(10);
  cout << "After constructing a.\n\n";

  cout << "Before call to display().\n";
  display(a);
  cout << "After display() returns.\n\n";

  cout << "Before call to mkBigger().\n";
  a = a.mkBigger();
  cout << "After mkBigger() returns.\n\n";

  cout << "Before second call to display().\n";
  display(a);
  cout << "After display() returns.\n\n";

  return 0;
}
```

mkBigger( ) returns a MyClass object.

The following output is produced:

```
Before constructing a.
Inside constructor
After constructing a.

Before call to display().
10
Destructing
After display() returns.

Before call to mkBigger().
Inside constructor
Destructing
Destructing
After mkBigger() returns.

Before second call to display().
20
Destructing
After display() returns.

Destructing
```

Notice the second "Destructing" message.

In this example, mkBigger( ) creates a local object called o that has a val value twice that of the invoking object. This object is then returned by the function and assigned to a inside main( ). Then o is destroyed, causing the first "Destructing" message to be displayed. But what explains the second call to the destructor?

When an object is returned by a function, a temporary object is automatically created, which 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. This is why the output shows a second "Destructing" message just before the message "After mkBigger( ) returns." This is the temporary object being destroyed.

As was the case when passing an object to a function, there is a potential problem when returning an object from a function. 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 releases a resource (such as memory or a file handle), that resource will be freed even though the object that is assigned the return value is still using it. The solution to this type of problem involves the use of a copy constructor, which is described next.

One last point: It is possible for a function to return an object by reference, but you need to be careful that the object being referenced does not go out of scope when the function is terminated.

1. Constructors cannot be overloaded. True or false?

2. When an object is passed by value to a function, a copy is made. Is this copy destroyed when the function returns?

3. When an object is returned by a function, a temporary object is created that contains the return value. True or false?

CRITICAL SKILL 9.5: Creating and Using a Copy Constructor

As earlier examples have shown, when an object is passed to or returned from a function, a copy of the object is made. By default, the copy is a bitwise clone of the original object. This default behavior is often acceptable, but in cases where it is not, you can control precisely how a copy of an object is made by explicitly defining a copy constructor for the class. A copy constructor is a special type of overloaded constructor that is automatically invoked when a copy of an object is required.

To begin, let's review why you might need to explicitly define a copy constructor. By default, when an object is passed to a function, a bitwise (that is, exact) copy of that object is made and given to the function parameter that receives the object. However, there are cases in which this identical copy is not desirable. For example, if the object uses a resource, such as an open file, then the copy will use the same resource as does the original object. Therefore, if the copy makes a change to that resource, it will be changed for the original object, too!

Furthermore, when the function terminates, the copy will be destroyed, thus causing its destructor to be called. This may cause the release of a resource that is still needed by the original object.

A similar situation occurs when an object is returned by a function. The compiler will generate a temporary object that holds a copy of the value returned by the function. (This is done automatically and is beyond your control.) This temporary object goes out of scope once the value is returned to the calling routine, causing the temporary object's destructor to be called. However, if the destructor destroys something needed by the calling code, trouble will follow.

At the core of these problems is the creation of a bitwise copy of the object. To prevent them, you need to define precisely what occurs when a copy of an object is made so that you can avoid undesired side effects. The way you accomplish this is by creating a copy constructor.

Before we explore the use of the copy constructor, it is important for you to understand that C++ defines two distinct types of situations in which the value of one object is given to another. The first situation is assignment. The second situation is initialization, which can occur 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. The copy constructor does not apply to assignments.

The most common form of copy constructor is shown here:

classname (const classname &obj) {

// body of constructor }

Here, obj is a reference to an object that is being used to initialize another object. For example, assuming a class called MyClass,and y as an object of type MyClass, then the following statements would invoke the MyClass copy constructor:

MyClass x = y; // y explicitly initializing x func1(y); // y passed as a parameter y = func2(); // y receiving a returned object

In the first two cases, a reference to y would be passed to the copy constructor. In the third, a reference to the object returned by func2( ) would be passed to the copy constructor. Thus, when an object is passed as a parameter, returned by a function, or used in an initialization, the copy constructor is called to duplicate the object.

Remember, the copy constructor is not called when one object is assigned to another. For example, the following sequence will not invoke the copy constructor:

MyClass x; MyClass y;

x = y; // copy constructor not used here.

Again, assignments are handled by the assignment operator, not the copy constructor.

The following program demonstrates a copy constructor:

```
/* Copy constructor invoked when passing an object
   to a function. */

#include <iostream>
using namespace std;

class MyClass {
  int val;
  int copynumber;
public:
  // Normal constructor.
  MyClass(int i) {
    val = i;
    copynumber = 0;
    cout << "Inside normal constructor\n";
  }

  // Copy constructor
  MyClass(const MyClass &o) {
    val = o.val;
    copynumber = o.copynumber + 1;
    cout << "Inside copy constructor.\n";
  }

  ~MyClass() {
     if(copynumber == 0)
       cout << "Destructing original.\n";
     else
       cout << "Destructing copy " <<
                copynumber << "\n";
  }

  int getval() { return val; }
};

void display(MyClass ob)
{
  cout << ob.getval() << '\n';
}

int main()
{
  MyClass a(10);
  display(a);

  return 0;
}
```

This is the **MyClass** copy constructor.

The copy constructor is called when **a** is passed to **display( )**.

This program displays the following output:

```
Inside normal constructor
Inside copy constructor.
10
Destructing copy 1
Destructing original.
```

Here is what occurs when the program is run: When a is created inside main( ), the value of its copynumber is set to 0 by the normal constructor. Next, a is passed to ob of display( ). When this occurs, the copy constructor is called, and a copy of a is created. In the process, the copy constructor increments the value of copynumber. When display( ) returns, ob goes out of scope. This causes its destructor to be called. Finally, when main( ) returns, a goes out of scope.

You might want to try experimenting with the preceding program a bit. For example, create a function that returns a MyClass object, and observe when the copy constructor is called.

1. When the default copy constructor is used, how is a copy of an object made?
2. A copy constructor is called when one object is assigned to another. True or false?
3. Why might you need to explicitly define a copy constructor for a class?

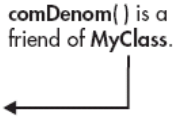
CRITICAL SKILL 9.6: Friend Functions

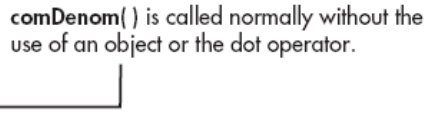
In general, only other members of a class have access to the private members of the class. However, it is possible to allow a nonmember function access to the private members of a class by declaring it as a friend of the class. To make a function a friend of a class, you include its prototype in the public section of the class declaration and precede it with the friend keyword. For example, in this fragment, frnd( ) is declared to be a friend of the class MyClass:

```
class MyClass { // ... public: friend void frnd(MyClass ob); // ... };
```

As you can see, the keyword friend precedes the rest of the prototype. A function can be a friend of more than one class. Here is a short example that uses a friend function to determine if the private fields of MyClass have a common denominator:

```
// Demonstrate a friend function.

#include <iostream>
using namespace std;

class MyClass {
  int a, b;
public:
  MyClass(int i, int j) { a=i; b=j; }
  friend int comDenom(MyClass x); // a friend function
};

// Notice that comDenom() is a not a member function of any class.
int comDenom(MyClass x)
{
  /* Because comDenom() is a friend of MyClass, it can
     directly access a and b. */
  int max = x.a < x.b ? x.a : x.b;

  for(int i=2; i <= max; i++)
    if((x.a%i)==0 && (x.b%i)==0) return i;

  return 0;
}

int main()
{
  MyClass n(18, 111);

  if(comDenom(n))
    cout << "Common denominator is " <<
            comDenom(n) << "\n";
  else
    cout << "No common denominator.\n";

  return 0;
}
```





In this example, the comDenom( ) function is not a member of MyClass. However, it still has full access to the private members of MyClass. Specifically, it can access x.a and x.b. Notice also that comDenom( ) is called normally— that is, not in conjunction with an object and the dot operator. Since it is not a member function, it does not need to be qualified with an object's name. (In fact, it cannot be qualified with an object.) Typically, a friend function is passed one or more objects of the class for which it is a friend, as is the case with comDenom( ).

While there is nothing gained by making comDenom( ) a friend rather than a member function of MyClass, there are some circumstances in which friend functions are quite valuable. First, friends can be useful for overloading certain types of operators, as described later in this module. Second, friend functions simplify the creation of some types of I/O functions, as described in Module 11.

The third reason that friend functions may be desirable is that, in some cases, two or more classes can contain members that are interrelated relative to other parts of your program. For example, imagine

two different classes called Cube and Cylinder that define the characteristics of a cube and cylinder, of which one of these characteristics is the color of the object. To enable the color of a cube and cylinder to be easily compared, you can define a friend function that compares the color component of each object, returning true if the colors match and false if they differ. The following program illustrates this concept:

```
// Friend functions can be shared by two or more classes.

#include <iostream>
using namespace std;

class Cylinder; // a forward declaration

enum colors { red, green, yellow };

class Cube {
 colors color;
public:
  Cube(colors c) { color = c; }
  friend bool sameColor(Cube x, Cylinder y);
  // ...
};
class Cylinder {
 colors color;
public:
  Cylinder(colors c) { color= c; }
  friend bool sameColor(Cube x, Cylinder y);
  // ...
};

bool sameColor(Cube x, Cylinder y)
{
  if(x.color == y.color) return true;
  else return false;
}

int main()
{
  Cube cube1(red);
  Cube cube2(green);
  Cylinder cyl(green);


  if(sameColor(cube1, cyl))
    cout << "cube1 and cyl are the same color.\n";
  else
    cout << "cube1 and cyl are different colors.\n";

  if(sameColor(cube2, cyl))
    cout << "cube2 and cyl are the same color.\n";
  else
    cout << "cube2 and cyl are different colors.\n";

  return 0;
}
```

sameColor( ) is a friend of **Cube**.

sameColor( ) is also a friend of **Cylinder**.

The output produced by this program is shown here:

```
cube1 and cyl are different colors.

cube2 and cyl are the same color.
```

Notice that this program uses a forward declaration (also called a forward reference) for the class Cylinder. This is necessary because the declaration of sameColor( ) inside Cube refers to Cylinder before it is declared. To create a forward declaration to a class, simply use the form shown in this program.

A friend of one class can be a member of another. For example, here is the preceding program rewritten so that sameColor( ) is a member of Cube. Notice the use of the scope resolution operator when declaring sameColor( ) to be a friend of Cylinder.

```
/* A function can be a member of one class and
   a friend of another. */

#include <iostream>
using namespace std;

class Cylinder; // a forward declaration

enum colors { red, green, yellow };

class Cube {
 colors color;
public:
  Cube(colors c) { color= c; }
  bool sameColor(Cylinder y); <-- sameColor( ) is now a member of Cube.
  // ...
};

class Cylinder {
 colors color;
public:
  Cylinder(colors c) { color = c; }
  friend bool Cube::sameColor(Cylinder y); <-- Cube::sameColor( ) is a friend of Cylinder.
  // ...
};

bool Cube::sameColor(Cylinder y) {
  if(color == y.color) return true;
  else return false;
}

int main()
{
  Cube cube1(red);
  Cube cube2(green);
  Cylinder cyl(green);

  if(cube1.sameColor(cyl))
    cout << "cube1 and cyl are the same color.\n";
  else
    cout << "cube1 and cyl are different colors.\n";

  if(cube2.sameColor(cyl))
    cout << "cube2 and cyl are the same color.\n";
  else
    cout << "cube2 and cyl are different colors.\n";

  return 0;
}
```

Since sameColor( ) is a member of Cube, it must be called on a Cube object, which means that it can access the color variable of objects of type Cube directly. Thus, only objects of type Cylinder need to be passed to sameColor( ).

1. What is a friend function? What keyword declares one?
2. Is a friend function called on an object using the dot operator?
3. Can a friend of one class be a member of another?

CRITICAL SKILL 9.7: Structures and Unions

In addition to the keyword class, C++ gives you two other ways to create a class type. First, you can create a structure. Second, you can create a union. Each is examined here.

Structures

Structures are inherited from the C language and are declared using the keyword struct. A struct is syntactically similar to a class, and both create a class type. In the C language, a struct can contain only data members, but this limitation does not apply to C++. In C++, the struct is essentially just an alternative way to specify a class. In fact, in C++ the only difference between a class and a struct is that by default all members are public in a struct and private in a class. In all other respects, structures and classes are equivalent.

Here is an example of a structure:

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

struct Test {
  int get_i() { return i; } // these are public
  void put_i(int j) { i = j; } // by default
private:
  int i;
};

int main()
{
  Test s;

  s.put_i(10);
  cout << s.get_i();

  return 0;
}
```

Structure members are public by default.

This simple program defines a structure type called Test, in which get_i( ) and put_i( ) are public and i is private. Notice the use of the keyword private to specify the private elements of the structure.

The following program shows an equivalent program that uses a class instead of a struct:

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

class Test {
  int i; // private by default
public:
  int get_i() { return i; }
  void put_i(int j) { i = j; }
};

int main()
{
  Test s;

  s.put_i(10);
  cout << s.get_i();

  return 0;
}
```

Ask the Expert

Q: Since struct and class are so similar, why does C++ have both?

A: On the surface, there is seeming redundancy in the fact that both structures and classes have virtually identical capabilities. Many newcomers to C++ wonder why this apparent duplication exists. In fact, it is not uncommon to hear the suggestion that either the keyword class or struct is unnecessary.

The answer to this line of reasoning is rooted in the desire to keep C++ compatible with C. As C++ is currently defined, a standard C structure is also a completely valid C++ structure. In C, which has no concept of public or private structure members, all structure members are public by default. This is why members of C++ structures are public (rather than private) by default. Since the class keyword is expressly designed to support encapsulation, it makes sense that its members are private by default. Thus, to avoid incompatibility with C on this issue, the structure default could not be altered, so a new keyword was added. However, in the long term, there is a more important reason for the separation of structures and classes. Because class is an entity syntactically separate from struct, the definition of a class is free to evolve in ways that may not be syntactically compatible with C-like structures. Since the two are separated, the future direction of C++ will not be encumbered by concerns of compatibility with C-like structures.

For the most part, C++ programmers will use a class to define the form of an object that contains member functions and will use a struct in its more traditional role to create objects that contain only data members. Sometimes the acronym "POD" is used to describe a structure that does not contain member functions. It stands for "plain old data."

Unions

A union is a memory location that is shared by two or more different variables. A union is created using the keyword union, and its declaration is similar to that of a structure, as shown in this example:

```
union utype { short int i; char ch;

} ;
```

This defines a union in which a short int value and a char value share the same location. Be clear on one point: It is not possible to have this union hold both an integer and a character at the same time, because i and ch overlay each other. Instead, your program can treat the information in the union as an integer or as a character at any time. Thus, a union gives you two or more ways to view the same piece of data.

You can declare a union variable by placing its name at the end of the union declaration, or by using a separate declaration statement. For example, to declare a union variable called u_var of type utype, you would write

```
utype u_var;
```

In u_var, both the short integer i and the character ch share the same memory location. (Of course, i occupies two bytes and ch uses only one.) Figure 9-1 shows how i and ch both share the same address.

As far as C++ is concerned, a union is essentially a class in which all elements are stored in the same location. In fact, a union defines a class type. A union can contain constructors and destructors as well as member functions. Because the union is inherited from C, its members are public, not private, by default.

Here is a program that uses a union to display the characters that comprise the low- and high-order bytes of a short integer (assuming short integers are two bytes):

```
// Demonstrate a union.

#include <iostream>
using namespace std;

union u_type {
  u_type(short int a) { i = a; };
  u_type(char x, char y) { ch[0] = x; ch[1] = y; }

  void showchars(){
    cout << ch[0] << " ";
    cout << ch[1] << "\n";
  }

  short int i;
  char ch[2];
};

int main()
{
  u_type u(1000);
  u_type u2('X', 'Y');

  cout << "u as integer: ";
  cout << u.i << "\n";
  cout << "u as chars: ";
  u.showchars();

  cout << "u2 as integer: ";
  cout << u2.i << "\n";
  cout << "u2 as chars: ";
  u2.showchars();

  return 0;
}
```

Union data members share the same memory.

The data in a **u_type** object can be viewed as a short integer or as two characters.

The output is shown here:

```
u as integer: 1000
u as chars: è
u2 as integer: 22872
u2 as chars: X Y
```

As the output shows, using the u_type union, it is possible to view the same data two different ways.

Like the structure, the C++ union is derived from its C forerunner. However, in C, unions can include only data members; functions and constructors are not allowed. In C++, the union has the expanded capabilities of the class. But just because C++ gives unions greater power and flexibility does not mean that you have to use it. Often unions contain only data. However, in cases where you can encapsulate a union along with the routines that manipulate it, you will be adding considerable structure to your program by doing so.

There are several restrictions that must be observed when you use C++ unions. Most of these have to do with features of C++ that will be discussed later in this book, but they are mentioned here for completeness. First, a union cannot inherit a class. Further, a union cannot be a base class. A union cannot have virtual member functions. No static variables can be

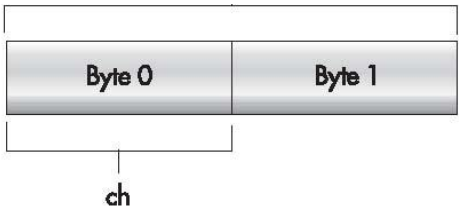


Figure 9-1 How **i** and **ch** of **u_var** both share the same address

members of a union. A reference member cannot be used. A union cannot have as a member any object that overloads the = operator. Finally, no object can be a member of a union if the object has an explicit constructor or destructor.

Anonymous Unions

There is a special type of union in C++ called an anonymous union. An anonymous union does not include a type name, and no variables of the union can be declared. Instead, an anonymous union tells the compiler that its member variables are to share the same location. However, the variables themselves are referred to directly, without the normal dot operator syntax. For example, consider this program:

```
// Demonstrate an anonymous union.

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

int main()
{
  // define anonymous union
  union {
    long l;
    double d;
    char s[4];
  } ;

  // now, reference union elements directly
  l = 100000;
  cout << l << " ";
  d = 123.2342;
  cout << d << " ";
  strcpy(s, "hi");
  cout << s;

  return 0;
}
```

This is an anonymous union.

The elements of an anonymous union are referred to directly.

As you can see, the elements of the union are referenced as if they had been declared as normal local variables. In fact, relative to your program, that is exactly how you will use them. Further, even though they are defined within a union declaration, they are at the same scope level as any other local variable within the same block. This implies that the names of the members of an anonymous union must not conflict with other identifiers known within the same scope.

All restrictions involving unions apply to anonymous ones, with these additions. First, the only elements contained within an anonymous union must be data. No member functions are allowed. Anonymous unions cannot contain private or protected elements. (The protected specifier is discussed in Module 10.) Finally, global anonymous unions must be specified as static.

CRITICAL SKILL 9.8: The this Keyword

Before moving on to operator overloading, it is necessary to describe another C++ keyword: this. Each time a member function is invoked, it is automatically passed a pointer, called this, to the object on which it is called. The this pointer is an implicit parameter to all member functions. Therefore, inside a member function, this can be used to refer to the invoking object.

As you know, a member function can directly access the private data of its class. For example, given this class:

```
class Test {
  int i;
  void f() { ... };
  // ...
};
```

inside f( ), the following statement can be used to assign i the value 10:

```
i = 10;
```

In actuality, the preceding statement is shorthand for this one:

```
this->i = 10;
```

To see the this pointer in action, examine the following short program:

```
// Use the "this" pointer.

#include <iostream>
using namespace std;

class Test {
  int i;
public:
  void load_i(int val) {
    this->i = val;  <-------- Same as i = val;
  }
  int get_i() {
    return this->i;  <-------- Same as return i;
  }
} ;

int main()
{
  Test o;

  o.load_i(100);
  cout << o.get_i();

  return 0;
}
```

This program displays the number 100. This example is, of course, trivial, and no one would actually use the this pointer in this way. Soon, however, you will see why the this pointer is important to C++ programming.

One other point: Friend functions do not have a this pointer, because friends are not members of a class. Only member functions have a this pointer.

1. Can a struct contain member functions?
2. What is the defining characteristic of a union?
3. To what does this refer?

CRITICAL SKILL 9.9: Operator Overloading

The remainder of this module explores one of C++'s most exciting and powerful features: operator overloading. In C++, operators can be overloaded relative to class types that you create. The principal advantage to overloading operators is that it allows you to seamlessly integrate new data types into your programming environment.

When you overload an operator, you define the meaning of an operator for a particular class. For example, a class that defines a linked list might use the + operator to add an object to the list. A class that implements a stack might use the + to push an object onto the stack.

Another class might use the + operator in an entirely different way. When an operator is overloaded, none of its original meaning is lost. It is simply that a new operation, relative to a specific class, is defined. Therefore, overloading the + to handle a linked list, for example, does not cause its meaning relative to integers (that is, addition) to be changed.

Operator overloading is closely related to function overloading. To overload an operator, you must define what the operation means relative to the class to which it is applied. To do this, you create an operator function. The general form of an operator function is

```
type classname::operator#(arg-list)
{ // operations
}
```

Here, the operator that you are overloading is substituted for the #, and type is the type of value returned by the specified operation. Although it can be of any type you choose, the return value is often of the same type as the class for which the operator is being overloaded. This correlation facilitates the use of the overloaded operator in compound expressions. Thespecificnatureof arg-list is determined by several factors, described in the sections that follow.

Operator functions can be either members or nonmembers of a class. Nonmember operator functions are often friend functions of the class, however. Although similar, there are some differences between the way a member operator function is overloaded and the way a nonmember operator function is overloaded. Each approach is described here.

NOTE: Because C++ defines many operators, the topic of operator overloading is quite large, and it is not possible to describe every aspect of it in this book. For a comprehensive description of operator overloading, refer to my book C++: The Complete Reference, Osborne/McGraw-Hill.

CRITICAL SKILL 9.10: Operator Overloading Using Member Functions

To begin our examination of member operator functions, let's start with a simple example. The following program creates a class called ThreeD, which maintains the coordinates of an object in three-dimensional space. This program overloads the + and the = operators relative to the ThreeD class. Examine it closely.

```
// Define + and = for the ThreeD class.

#include <iostream>
using namespace std;

class ThreeD {
  int x, y, z; // 3-D coordinates
public:
  ThreeD() { x = y = z = 0; }
  ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

  ThreeD operator+(ThreeD op2); // op1 is implied
  ThreeD operator=(ThreeD op2); // op1 is implied

  void show() ;
};

// Overload +.
ThreeD ThreeD::operator+(ThreeD op2) <-- Overload + for ThreeD.
{
  ThreeD temp;

  temp.x = x + op2.x; // These are integer additions
  temp.y = y + op2.y; // and the + retains its original
  temp.z = z + op2.z; // meaning relative to them.
  return temp; <-- Return a new object. Leave arguments unchanged.
}

// Overload assignment.
ThreeD ThreeD::operator=(ThreeD op2) <-- Overload = for ThreeD.
{
  x = op2.x; // These are integer assignments
  y = op2.y; // and the = retains its original
  z = op2.z; // meaning relative to them.
  return *this; <-- Return the modified object.
}
```

```
}

// Show X, Y, Z coordinates.
void ThreeD::show()
{
  cout << x << ", ";
  cout << y << ", ";
  cout << z << "\n";
}

int main()
{
  ThreeD a(1, 2, 3), b(10, 10, 10), c;

  cout << "Original value of a: ";
  a.show();
  cout << "Original value of b: ";
  b.show();

  cout << "\n";

  c = a + b; // add a and b together
  cout << "Value of c after c = a + b: ";
  c.show();

  cout << "\n";

  c = a + b + c; // add a, b and c together
  cout << "Value of c after c = a + b + c: ";
  c.show();

  cout << "\n";

  c = b = a;  // demonstrate multiple assignment
  cout << "Value of c after c = b = a: ";
  c.show();
  cout << "Value of b after c = b = a: ";
  b.show();

  return 0;
}
```

This program produces the following output:

```
Original value of a: 1, 2, 3
Original value of b: 10, 10, 10
Value of c after c = a + b: 11, 12, 13
Value of c after c = a + b + c: 22, 24, 26
Value of c after c = b = a: 1, 2, 3
```

```
Value of b after c = b = a: 1, 2, 3
```

As you examined the program, you may have been surprised to see that both operator functions have only one parameter each, even though they overload binary operations. The reason for this apparent contradiction is that when a binary operator is overloaded using a member function, only one argument is explicitly passed to it. The other argument is implicitly passed using the this pointer. Thus, in the line

```
temp.x = x + op2.x;
```

the x refers to this–>x, which is the x associated with the object that invokes the operator function. In all cases, it is the object on the left side of an operation that causes the call to the operator function. The object on the right side is passed to the function.

In general, when you use a member function, no parameters are used when overloading a unary operator, and only one parameter is required when overloading a binary operator. (You cannot overload the ternary ? operator.) In either case, the object that invokes the operator function is implicitly passed via the this pointer.

To understand how operator overloading works, let's examine the preceding program carefully, beginning with the overloaded operator +. When two objects of type ThreeD are operated on by the + operator, the magnitudes of their respective coordinates are added together, as shown in operator+( ). Notice, however, that this function does not modify the value of either operand. Instead, an object of type ThreeD, which contains the result of the operation, is returned by the function. To understand why the + operation does not change the contents of either object, think about the standard arithmetic + operation as applied like this: 10 + 12. The outcome of this operation is 22, but neither 10 nor 12 is changed by it. Although there is no rule that prevents an overloaded + operator from altering the value of one of its operands, it is best for the actions of an overloaded operator to be consistent with its original meaning.

Notice that operator+( ) returns an object of type ThreeD. Although the function could have returned any valid C++ type, the fact that it returns a ThreeD object allows the + operator to be used in compound expressions, such as a+b+c. Here, a+b generates a result that is of type ThreeD. This value can then be added to c. Had any other type of value been generated by a+b, such an expression would not work.

In contrast with the + operator, the assignment operator does, indeed, cause one of its arguments to be modified. (This is, after all, the very essence of assignment.) Since the operator=( ) function is called by the object that occurs on the left side of the assignment, it is this object that is modified by the assignment operation. Most often, the return value of an overloaded assignment operator is the object on the left, after the assignment has been made.

(This is in keeping with the traditional action of the = operator.) For example, to allow statements like

```
a = b = c = d;
```

it is necessary for operator=( ) to return the object pointed to by this, which will be the object that occurs on the left side of the assignment statement. This allows a chain of assignments to be made. The assignment operation is one of the most important uses of the this pointer.

In the preceding program, it was not actually necessary to overload the = because the default assignment operator provided by C++ is adequate for the ThreeD class. (As explained earlier in this module, the default assignment operation is a bitwise copy.) The = was overloaded simply to show the proper procedure. In general, you need to overload the = only when the default bitwise copy cannot be used. Because the default = operator is sufficient for ThreeD, subsequent examples in this module will not overload it.

Order Matters

When overloading binary operators, remember that in many cases, the order of the operands does make a difference. For example, although A + B is commutative, A – B is not. (That is, A – B is not the same as B – A!) Therefore, when implementing overloaded versions of the noncommutative operators, you must remember which operand is on the left and which is on the right. For example, here is how to overload the minus for the ThreeD class:

```
// Overload subtraction.
ThreeD ThreeD::operator-(ThreeD op2)
{
  ThreeD temp;

  temp.x = x - op2.x;
  temp.y = y - op2.y;
  temp.z = z - op2.z;
  return temp;
}
```

Remember, it is the operand on the left that invokes the operator function. The operand on the right is passed explicitly.

Using Member Functions to Overload Unary Operators

You can also overload unary operators, such as ++, – –, or the unary – or +. As stated earlier, when a unary operator is overloaded by means of a member function, no object is explicitly passed to the operator function. Instead, the operation is performed on the object that generates the call to the function through the implicitly passed this pointer. For example, here is a program that defines the increment operation for objects of type ThreeD:

```
// Overload the ++ unary operator.

#include <iostream>
using namespace std;

class ThreeD {
  int x, y, z; // 3-D coordinates
public:
  ThreeD() { x = y = z = 0; }
  ThreeD(int i, int j, int k) {x = i; y = j; z = k; }

  ThreeD operator++(); // prefix version of ++

  void show() ;
} ;

// Overload the prefix version of ++.
ThreeD ThreeD::operator++() <-------- Overload ++ for ThreeD.
{
  x++; // increment x, y, and z
  y++;
  z++;
  return *this; <-------- Return the incremented object.
}

// Show X, Y, Z coordinates.
void ThreeD::show()
{
  cout << x << ", ";
  cout << y << ", ";
  cout << z << "\n";
}

int main()
{
  ThreeD a(1, 2, 3);

  cout << "Original value of a: ";
  a.show();

  ++a;  // increment a
  cout << "Value after ++a: ";
  a.show();

  return 0;
}
```

The output is shown here:

```
Original value of a: 1, 2, 3

Value after ++a: 2, 3, 4
```

As the output verifies, operator++( ) increments each coordinate in the object and returns the modified object. Again, this is in keeping with the traditional meaning of the ++ operator. As you know, the ++ and – – have both a prefix and a postfix form. For example, both

++x;

and

x++;

are valid uses of the increment operator. As the comments in the preceding program state, the operator++( ) function defines the prefix form of ++ relative to the ThreeD class. However, it is possible to overload the postfix form as well. The prototype for the postfix form of the ++ operator relative to the ThreeD class is shown here:

ThreeD operator++(int notused);

The parameter notused is not used by the function and should be ignored. This parameter is simply a way for the compiler to distinguish between the prefix and postfix forms of the increment operator. (The postfix decrement uses the same approach.)

Here is one way to implement a postfix version of ++ relative to the ThreeD class:

```
// Overload the postfix version of ++.
ThreeD ThreeD::operator++(int notused)  <-- Notice the notused parameter.
{
  ThreeD temp = *this; // save original value
  x++;  // increment x, y, and z
  y++;
  z++;
  return temp; // return original value
}
```

Notice that this function saves the current state of the operand using the statement

ThreeD temp = *this;

and then returns temp. Keep in mind that the normal meaning of a postfix increment is to first obtain the value of the operand, and then to increment the operand. Therefore, it is necessary to save the current state of the operand and return its original value, before it is incremented, rather than its modified value.

The following program implements both forms of the ++ operator:

```
// Demonstrate prefix and postfix ++.

#include <iostream>
using namespace std;

class ThreeD {
  int x, y, z; // 3-D coordinates
public:
  ThreeD() { x = y = z = 0; }
  ThreeD(int i, int j, int k) {x = i; y = j; z = k; }

  ThreeD operator++(); // prefix version of ++
  ThreeD operator++(int notused); // postfix version of ++

  void show() ;
};

// Overload the prefix version of ++.
ThreeD ThreeD::operator++()
{
  x++;  // increment x, y, and z
  y++;
  z++;
  return *this; // return altered value
}

// Overload the postfix version of ++.
ThreeD ThreeD::operator++(int notused)
{
  ThreeD temp = *this; // save original value

  x++; // increment x, y, and z
  y++;
  z++;
  return temp; // return original value
}

// Show X, Y, Z coordinates.
void ThreeD::show( )
{
  cout << x << ", ";
  cout << y << ", ";
  cout << z << "\n";
}
```

```
int main()
{
  ThreeD a(1, 2, 3);
  ThreeD b;

  cout << "Original value of a: ";
  a.show();

  cout << "\n";

  ++a; // prefix increment
  cout << "Value after ++a: ";
  a.show();

  a++; // postfix increment
  cout << "Value after a++: ";
  a.show();

  cout << "\n";

  b = ++a; // b receives a's value after increment
  cout << "Value of a after b = ++a: ";
  a.show();
  cout << "Value of b after b = ++a: ";
  b.show();

  cout << "\n";

  b = a++; // b receives a's value prior to increment
  cout << "Value of a after b = a++: ";
  a.show();
  cout << "Value of b after b = a++: ";
  b.show();

  return 0;
}
```

Calls prefix increment function.

Calls postfix increment function.

The output from the program is shown here:

```
Original value of a: 1, 2, 3

Value after ++a: 2, 3, 4

Value after a++: 3, 4, 5

Value of a after b = ++a: 4, 5, 6

Value of b after b = ++a: 4, 5, 6

Value of a after b = a++: 5, 6, 7

Value of b after b = a++: 4, 5, 6
```

Remember that if the ++ precedes its operand, the operator++( ) is called. If it follows its operand, the operator++(int notused) function is called. This same approach is also used to overload the prefix and postfix decrement operator relative to any class. You might want to try defining the decrement operator relative to ThreeD as an exercise.

As a point of interest, early versions of C++ did not distinguish between the prefix and postfix forms of the increment or decrement operators. For these old versions, the prefix form of the operator function was called for both uses of the operator. When working on older C++ code, be aware of this possibility.

1. Operators must be overloaded relative to a class. True or false?
2. How many parameters does a member operator function have for a binary operator?
3. For a binary member operator function, the left operand is passed via _____.

CRITICAL SKILL 9.11: Nonmember Operator Functions

You can overload an operator for a class by using a nonmember function, which is often a friend of the class. As you learned earlier, friend functions do not have a this pointer. Therefore, when a friend is used to overload an operator, both operands are passed explicitly when a binary operator is overloaded, and one operand is passed explicitly when a unary operator is overloaded. The only operators that cannot be overloaded using friend functions are =, (), [ ], and –>.

The following program uses a friend instead of a member function to overload the + operator for the ThreeD class:

```
// Use friend operator functions.

#include <iostream>
using namespace std;

class ThreeD {
  int x, y, z; // 3-D coordinates
public:
  ThreeD() { x = y = z = 0; }
  ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

  friend ThreeD operator+(ThreeD op1, ThreeD op2);

  void show() ;
} ;

// The + is now a friend function.
ThreeD operator+(ThreeD op1, ThreeD op2)
{
  ThreeD temp;

  temp.x = op1.x + op2.x;
  temp.y = op1.y + op2.y;
  temp.z = op1.z+ op2.z;
  return temp;
}

// Show X, Y, Z coordinates.
void ThreeD::show()
{
  cout << x << ", ";
  cout << y << ", ";
```

Here, **operator+( )** is a friend of **ThreeD**. Notice that two parameters are required.

```
  cout << z << "\n";
}

int main()
{
  ThreeD a(1, 2, 3), b(10, 10, 10), c;

  cout << "Original value of a: ";
  a.show();
  cout << "Original value of b: ";
  b.show();

  cout << "\n";

  c = a + b; // add a and b together
  cout << "Value of c after c = a + b: ";
  c.show();

  cout << "\n";

  c = a + b + c; // add a, b and c together
  cout << "Value of c after c = a + b + c: ";
  c.show();

  cout << "\n";

  c = b = a;  // demonstrate multiple assignment
  cout << "Value of c after c = b = a: ";
  c.show();
  cout << "Value of b after c = b = a: ";
  b.show();

  return 0;
}
```

The output is shown here:

```
Original value of a: 1, 2, 3

Original value of b: 10, 10, 10

Value of c after c = a + b: 11, 12, 13

Value of c after c = a + b + c: 22, 24, 26

Value of c after c = b = a: 1, 2, 3

Value of b after c = b = a: 1, 2, 3
```

As you can see by looking at operator+( ), now both operands are passed to it. The left operand is passed in op1, and the right operand in op2.

In many cases, there is no benefit to using a friend function instead of a member function when overloading an operator. However, there is one situation in which a friend function is quite useful: when you want an object of a built-in type to occur on the left side of a binary operation. To understand why, consider the following. As you know, a pointer to the object that invokes a member operator function is passed in this. In the case of a binary operator, it is the object on the left that invokes the function. This is fine, provided that the object on the left defines the specified operation. For example, assuming some object called T, which has assignment and integer addition defined for it, then this is a perfectly valid statement:

```
T = T + 10; // will work
```

Since the object T is on the left side of the + operator, it invokes its overloaded operator function, which (presumably) is capable of adding an integer value to some element of T. However, this statement won't work:

```
T = 10 + T; // won't work
```

The problem with this statement is that the object on the left of the + operator is an integer, a built-in type for which no operation involving an integer and an object of T's type is defined. The solution to the preceding problem is to overload the + using two friend functions. In

this case, the operator function is explicitly passed both arguments and is invoked like any other overloaded function, based upon the types of its arguments. One version of the + operator function handles object + integer, and the other handles integer + object. Overloading the + (or any other binary operator) using friend functions allows a built-in type to occur on the left or right side of the operator. The following program illustrates this technique. It defines two versions of operator+( ) for objects of type ThreeD. Both add an integer value to each of ThreeD's instance variables. The integer can be on either the left or right side of the operator.

```
// Overload for integer + object and object + integer.

#include <iostream>
using namespace std;

class ThreeD {
  int x, y, z; // 3-D coordinates
public:
  ThreeD() { x = y = z = 0; }
  ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

  friend ThreeD operator+(ThreeD op1, int op2);
  friend ThreeD operator+(int op1, ThreeD op2);
```

These allow ob + int and int + ob.

```
  void show() ;
} ;

// This allows ThreeD + int
ThreeD operator+(ThreeD op1, int op2)
{
  ThreeD temp;

  temp.x = op1.x + op2;
  temp.y = op1.y + op2;
  temp.z = op1.z + op2;
  return temp;
}

// This allows int + ThreeD
ThreeD operator+(int op1, ThreeD op2)
{
  ThreeD temp;

  temp.x = op2.x + op1;
  temp.y = op2.y + op1;
  temp.z = op2.z + op1;
  return temp;
}

// Show X, Y, Z coordinates.
void ThreeD::show()
{
  cout << x << ", ";
  cout << y << ", ";
  cout << z << "\n";
}

int main()
{
  ThreeD a(1, 2, 3), b;

  cout << "Original value of a: ";
  a.show();

  cout << "\n";

  b = a + 10; // object + integer
  cout << "Value of b after b = a + 10: ";
  b.show();
```

```
  cout << "\n";

  b = 10 + a; // integer + object
  cout << "Value of b after b = 10 + a: ";
  b.show();

  return 0;
}
```

Here, the built-in type occurs on the left side of an addition.

The output is shown here:

Original value of a: 1, 2, 3

Value of b after b = a + 10: 11, 12, 13

Value of b after b = 10 + a: 11, 12, 13

Because the operator+( ) function is overloaded twice, it can accommodate the two ways in which an integer and an object of type ThreeD can occur in the addition operation.

Using a Friend to Overload a Unary Operator

You can also overload a unary operator by using a friend function. However, if you are overloading the ++ or – –, you must pass the operand to the function as a reference parameter. Since a reference parameter is an implicit pointer to the argument, changes to the parameter will affect the argument. Using a reference parameter allows the function to increment or decrement the object used as an operand. When a friend is used for overloading the increment or decrement operators, the prefix form takes one parameter (which is the operand). The postfix form takes two parameters. The second parameter is an integer, which is not used. Here is the way to overload both forms of a friend operator++( ) function for the ThreeD class:

```
/* Overload prefix ++ using a friend function.
   This requires the use of a reference parameter. */
ThreeD operator++(ThreeD &op1)
{
  op1.x++;
  op1.y++;
  op1.z++;
  return op1;
}
```

```
/* Overload postfix ++ using a friend function.
   This requires the use of a reference parameter. */
ThreeD operator++(ThreeD &op1, int notused)
{
  ThreeD temp = op1;

  op1.x++;
  op1.y++;
  op1.z++;
  return temp;
}
```

1. How many parameters does a nonmember binary operator function have?

2. When using a nonmember operator function to overload the ++ operator, how must the operand be passed?

3. One advantage to using friend operator functions is that it allows a built-in type (such as int) to be used as the left operand. True or false?

Operator Overloading Tips and Restrictions

The action of an overloaded operator as applied to the class for which it is defined need not bear any relationship to that operator's default usage, as applied to C++'s built-in types. For example, the << and >> operators, as applied to cout and cin, have little in common with the same operators applied to integer types. However, for the purposes of the structure and readability of your code, an overloaded operator should reflect, when possible, the spirit of the operator's original use. For example, the + relative to ThreeD is conceptually similar to the + relative to integer types. There would be little benefit in defining the + operator relative to some class in such a way that it acts more the way you would expect the || operator, for instance, to perform. The central concept here is that although you can give an overloaded operator any meaning you like, for clarity it is best when its new meaning is related to its original meaning.

Ask the Expert

Q: Are there any special issues to consider when overloading the relational operators?

A: Overloading a relational operator, such as == or <, is a straightforward process. However, there is one small issue. As you know, an overloaded operator function often returns an object of the class for which it is overloaded. However, an overloaded relational operator typically returns true or false. This is

in keeping with the normal usage of relational operators and allows the overloaded relational operators to be used in conditional expressions. The same rationale applies when overloading the logical operators.

To show you how an overloaded relational operator can be implemented, the following function overloads == relative to the ThreeD class:

```
// Overload ==.
bool ThreeD::operator==(ThreeD op2)
{
  if((x == op2.x) && (y == op2.y) && (z == op2.z))
    return true;
  else
    return false;
}
```

Once operator==( ) has been implemented, the following fragment is perfectly valid:

```
ThreeD a(1, 1, 1), b(2, 2, 2); // ... if(a == b) cout << "a equals b\n"; else cout << "a does not equal b\n";
```

There are some restrictions to overloading operators. First, you cannot alter the precedence of any operator. Second, you cannot alter the number of operands required by the operator, although your operator function could choose to ignore an operand. Finally, except for the function call operator, operator functions cannot have default arguments.

Nearly all of the C++ operators can be overloaded. This includes specialized operators, such as the array indexing operator [ ], the function call operator (), and the –> operator. The only operators that you cannot overload are shown here:

. :: .* ?

The .* is a special-purpose operator whose use is beyond the scope of this book.

Project 9-1 Creating a Set Class

Operator overloading helps you create classes that can be fully integrated into the C++programming environment. Consider this point: by defining the necessary operators, you enable a class type to be used in a program in just the same way as you would use a built-in type. You can act on objects of that class through operators and use objects of that class in expressions. To illustrate the creation and integration of a new class into the C++ environment, this project creates a class called Set that defines a set type.

Before we begin, it is important to understand precisely what we mean by a set. For the purposes of this project, a set is a collection of unique elements. That is, no two elements in any given set can be the same. The ordering of a set's members is irrelevant. Thus, the set

{ A, B, C }

is the same as the set

{ A, C, B }

A set can also be empty.

Sets support a number of operations. The ones that we will implement are

- Adding an element to a set
- Removing an element from a set
- Set union
- Set difference

Adding an element to a set and removing an element from a set are self- explanatory operations. The other two warrant some explanation. The union of two sets is a set that contains all of the elements from both sets. (Of course, no duplicate elements are allowed.) We will use the + operator to perform a set union.

The difference between two sets is a set that contains those elements in the first set that are not part of the second set. We will use the – operator to perform a set difference. For example, given two sets S1 and S2, this statement removes the elements of S2 from S1, putting the result in S3:

S3 = S1 – S2

If S1 and S2 are the same, then S3 will be the null set. The Set class will also include a function called isMember( ), which determines if a specified element is a member of a given set. Of course, there are several other operations that can be performed on sets. Some are developed in the Mastery Check. Others you might find fun to try adding on your own.

For the sake of simplicity, the Set class stores sets of characters, but the same basic principles could be used to create a Set class capable of storing other types of elements.

Step by Step

1. Create a new file called Set.cpp.

2. Begin creating Set by specifying its class declaration, as shown here:

```
const int MaxSize = 100;

class Set {
  int len; // number of members
  char members[MaxSize]; // this array holds the set

  /* The find() function is private because it
     is not used outside the Set class. */
  int find(char ch); // find an element

public:

  // Construct a null set.
  Set() { len = 0; }

  // Return the number of elements in the set.
  int getLength() { return len; }

  void showset(); // display the set
  bool isMember(char ch);  // check for membership

  Set operator +(char ch); // add an element
  Set operator -(char ch); // remove an element

  Set operator +(Set ob2); // set union
  Set operator -(Set ob2); // set difference
};
```

Each set is stored in a char array referred to by members. The number of members actually in the set is stored in len. The maximum size of a set is MaxSize, which is set to 100. (You can increase this value if you work with larger sets.)
The Set constructor creates a null set, which is a set with no members. There is no need to create any other constructors, or to define an explicit copy constructor for the Set class, because the default bitwise copy is sufficient. The getLength( ) function returns the value of len, which is the number of elements currently in the set.

3. Begin defining the member functions, starting with the private function find( ), as shown here:

```
/* Return the index of the element
   specified by ch, or -1 if not found. */
int Set::find(char ch) {
  int i;

  for(i=0; i < len; i++)
    if(members[i] == ch) return i;

  return -1;
}
```

This function determines if the element passed in ch is a member of the set. It returns the index of the element if it is found and –1 if the element is not part of the set. This function is private because it is not

used outside the Set class. As explained earlier in this book, member functions can be private to their class. A private member function can be called only by other member functions in the class.

4. Add the showset( ) function, as shown here:

```
// Show the set.
void Set::showset() {
  cout << "{ ";
  for(int i=0; i<len; i++)
    cout << members[i] << " ";

  cout << "}\n";
}
```

This function displays the contents of a set.

5. Add the isMember( ) function, shown here, which determines if a character is a member of a set:

```
/* Return true if ch is a member of the set.
   Return false otherwise. */
bool Set::isMember(char ch) {
  if(find(ch) != -1) return true;
  return false;
}
```

This function calls find( ) to determine if ch is a member of the invoking set. If it is, isMember( ) returns true. Otherwise, it returns false.

6. Begin adding the set operators, beginning with set addition. To do this, overload + for objects of type Set, as shown here. This version adds an element to a set.

```
// Add a unique element to a set.
Set Set::operator +(char ch) {
  Set newset;

  if(len == MaxSize) {
    cout << "Set is full.\n";
    return *this; // return existing set
  }

  newset = *this; // duplicate the existing set

  // see if element already exists
  if(find(ch) == -1) { // if not found, then add
    // add new element to new set
    newset.members[newset.len] = ch;
    newset.len++;
  }
  return newset; // return updated set
}
```

This function bears some close examination. First, a new set is created, which will hold the contents of the original set plus the character specified by ch. Before the character in ch is added, a check is made to see if there is enough room in the set to hold another character. If there is room for the new element, the original set is assigned to newset. Next, the find( ) function is called to determine if ch is already part of the set. If it is not, then ch is added and len is updated. In either case, newset is returned. Thus, the original set is untouched by this operation.

7. Overload – so that it removes an element from the set, as shown here:

```
// Remove an element from the set.
Set Set::operator -(char ch) {
  Set newset;
  int i = find(ch); // i will be -1 if element not found

  // copy and compress the remaining elements
  for(int j=0; j < len; j++)
    if(j != i) newset = newset + members[j];

  return newset;
}
```

This function starts by creating a new null set. Then, find( ) is called to determine the index of ch within the original set. Recall that find( ) returns –1 if ch is not a member. Next, the elements of the original set are added to the new set, except for the element whose index matches that returned by find( ). Thus, the resulting set contains all of the elements of the original set except for ch. If ch was not part of the original set to begin with, then the two sets are equivalent.

8. Overload the + and – again, as shown here. These versions implement set union and set difference.

```
// Set union.
Set Set::operator +(Set ob2) {
  Set newset = *this; // copy the first set

  // Add unique elements from second set.
  for(int i=0; i < ob2.len; i++)
    newset = newset + ob2.members[i];

  return newset; // return updated set
}

// Set difference.
Set Set::operator -(Set ob2) {
  Set newset = *this; // copy the first set

  // Subtract elements from second set.
  for(int i=0; i < ob2.len; i++)
    newset = newset - ob2.members[i];

  return newset; // return updated set
}
```

As you can see, these functions utilize the previously defined versions of the + and – operators to help perform their operations. In the case of set union, a new set is created that contains the elements of the first set. Then, the elements of the second set are added. Because the + operation only adds an element if it is not already part of the set, the resulting set is the union (without duplication) of the two sets. The set difference operator subtracts matching elements.

9. Here is the complete code for the Set class along with a main( ) function that demonstrates it:

```
/* Project 9-1
A set class for characters. */
```

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

const int MaxSize = 100;

class Set {
  int len; // number of members
  char members[MaxSize]; // this array holds the set

  /* The find() function is private because it
     is not used outside the Set class. */
  int find(char ch); // find an element

public:

  // Construct a null set.
  Set() { len = 0; }

  // Return the number of elements in the set.
  int getLength() { return len; }

  void showset(); // display the set
  bool isMember(char ch);  // check for membership

  Set operator +(char ch); // add an element
  Set operator -(char ch); // remove an element

  Set operator +(Set ob2); // set union
  Set operator -(Set ob2); // set difference
};

/* Return the index of the element
   specified by ch, or -1 if not found. */
int Set::find(char ch) {
  int i;

  for(i=0; i < len; i++)
    if(members[i] == ch) return i;

  return -1;
}

// Show the set.
void Set::showset() {
  cout << "{ ";
  for(int i=0; i<len; i++)
```

```
    cout << members[i] << " ";

  cout << "}\n";
}

/* Return true if ch is a member of the set.
   Return false otherwise. */
bool Set::isMember(char ch) {
  if(find(ch) != -1) return true;
  return false;
}

// Add a unique element to a set.
Set Set::operator +(char ch) {
  Set newset;

  if(len == MaxSize) {
    cout << "Set is full.\n";
    return *this; // return existing set
  }

  newset = *this; // duplicate the existing set

  // see if element already exists
  if(find(ch) == -1) { // if not found, then add
    // add new element to new set
    newset.members[newset.len] = ch;
    newset.len++;
  }
  return newset; // return updated set
}

// Remove an element from the set.
Set Set::operator -(char ch) {
  Set newset;
  int i = find(ch); // i will be -1 if element not found

  // copy and compress the remaining elements
  for(int j=0; j < len; j++)
    if(j != i) newset = newset + members[j];

  return newset;
}

// Set union.
```

```
Set Set::operator +(Set ob2) {
  Set newset = *this; // copy the first set

  // Add unique elements from second set.
  for(int i=0; i < ob2.len; i++)
    newset = newset + ob2.members[i];

  return newset; // return updated set
}

// Set difference.
Set Set::operator -(Set ob2) {
  Set newset = *this; // copy the first set

  // Subtract elements from second set.
  for(int i=0; i < ob2.len; i++)
    newset = newset - ob2.members[i];

  return newset; // return updated set
}

// Demonstrate the Set class.
int main() {
  // construct 10-element empty Set
  Set s1;
  Set s2;
  Set s3;

  s1 = s1 + 'A';
  s1 = s1 + 'B';
  s1 = s1 + 'C';

  cout << "s1 after adding A B C: ";
  s1.showset();

  cout << "\n";

  cout << "Testing for membership using isMember().\n";
  if(s1.isMember('B'))
    cout << "B is a member of s1.\n";
  else
    cout << "B is not a member of s1.\n";

  if(s1.isMember('T'))
    cout << "T is a member of s1.\n";
```

```
else
  cout << "T is not a member of s1.\n";

cout << "\n";

s1 = s1 - 'B';
cout << "s1 after s1 = s1 - 'B': ";
s1.showset();

s1 = s1 - 'A';
cout << "s1 after s1 = s1 - 'A': ";
s1.showset();

s1 = s1 - 'C';
cout << "s1 after a1 = s1 - 'C': ";
s1.showset();

cout << "\n";

s1 = s1 + 'A';
s1 = s1 + 'B';
s1 = s1 + 'C';
cout << "s1 after adding A B C: ";
s1.showset();

cout << "\n";

s2 = s2 + 'A';
s2 = s2 + 'X';
s2 = s2 + 'W';

cout << "s2 after adding A X W: ";
s2.showset();

cout << "\n";

s3 = s1 + s2;
cout << "s3 after s3 = s1 + s2: ";
s3.showset();

s3 = s3 - s1;
cout << "s3 after s3 - s1: ";
s3.showset();
```

```
  cout << "\n";

  cout << "s2 after s2 = s2 - s2: ";
  s2 = s2 - s2;  // clear s2
  s2.showset();

  cout << "\n";

  s2 = s2 + 'C'; // add ABC in reverse order
  s2 = s2 + 'B';
  s2 = s2 + 'A';

  cout << "s2 after adding C B A: ";
  s2.showset();

  return 0;
}
```

The output from this program is shown here:

```
s1 after adding A B C: { A B C }

Testing for membership using isMember().
B is a member of s1.
T is not a member of s1.

s1 after s1 = s1 - 'B': { A C }
s1 after s1 = s1 - 'A': { C }
s1 after a1 = s1 - 'C': { }

s1 after adding A B C: { A B C }

s2 after adding A X W: { A X W }

s3 after s3 = s1 + s2: { A B C X W }
s3 after s3 - s1: { X W }

s2 after s2 = s2 - s2: { }

s2 after adding C B A: { C B A }
```

Module 9 Mastery Check

1. What is a copy constructor and when is it called? Show the general form of a copy constructor.

2. Explain what happens when an object is returned by a function. Specifically, when is its destructor called?

3. Given this class:

```
class T {
  int i, j;
public:
  int sum() {
    return i + j;
  }
};
```

show how to rewrite sum( ) so that it uses this.

4. What is a structure? What is a union?

5. Inside a member function, to what does *this refer?

6. What is a friend function?

7. Show the general form used for overloading a binary member operator function.

8. To allow operations involving a class type and a built-in type, what must you do?

9. Can the ? be overloaded? Can you change the precedence of an operator?

10. For the Set class developed in Project 9-1, define < and > so that they determine if one set is a subset or a superset of another set. Have < return true if the left set is a subset of the set on the right, and false otherwise. Have > return true if the left set is a superset of the set on the right, and false otherwise.

11. For the Set class, define the & so that it yields the intersection of two sets.

12. On your own, try adding other Set operators. For example, try defining | so that it yields the symmetric difference between two sets. The symmetric difference consists of those elements that the two sets do not have in common.