2.7 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 by 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 illustratesthis point:

```
#include <iostream.h>
class OBJ
{
   int i;
   public:
      void set_i(int x) { i = x; }
      void out_i() { cout<< i << " "; }
};
void f(OBJ x)
{
   x.out_i();// outputs 10
   x.set_i(100); // this affects only local
copy
   x.out_i();// outputs 100
}

int main()
{
   OBJ o;
   o.set_i(10);
   f(o);
   o.out_i();// still outputs 10, value of i
unchanged
   return 0;
}
```

The output from the program is shown here.

```
10 100 10
```

As the output shows, the modification of x within f( ) has no effect on object o insidemain( ).

2.8 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.h>
   class myclass
   {
intval;
   public:
     myclass(int    i){    val    =    i;    cout<<
   "Constructing\n"; }
     ~myclass() { cout<< "Destructing\n"; }
     intgetval() { return val; }
   };
   void display(myclassob)
   {
       cout<<ob.getval() << '\n';
   }
   int main()
   {
     myclass a(10);
     display(a);
     return 0;
   }
```

This program produces the following, unexpected output:

```
Constructing
10
Destructing
Destructing
```

As you can see, there is one call to the constructor function (which occurs when ais 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 (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. 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 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 bygoing out of scope when the function returns), the destructor is called.

2.9 A Potential Problem When Passing Objects

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 dynamic 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 is a problembecause the original object is still using the memory. This situation will leave the original object damaged and effectively useless. Consider this sample program:

```
// Demonstrate a problem when passing objects.
#include <iostream.h>
#include <stdlib.h>
class myclass
{
int *p;

public:
   myclass(int i);
   ~myclass();
   intgetval() { return *p; }
};
myclass::myclass(int i)
{
   cout<< "Allocating p\n";
   p = new int;
   *p = i;
}
myclass::~myclass()
{
cout<< "Freeing p\n";
  delete p;
}
// This will cause a problem.

void display(myclassob)
```

```
{
     cout<<ob.getval() << '\n';
}
int main()
{
myclass a(10);
display(a);
return 0;
}
```

This program displays the following output:

```
Allocating p
10
Freeing p
Freeing p
```

This program contains a fundamental error. Here is why: When **a** is constructed withinmain( ), memory is allocated and assigned to a.p. When **a** is passed to display( ),**a** is copied into the parameter ob. This means that both a and ob will have the same value for p. That is, both objects will have their copies of p pointing to the same dynamically allocated memory. When display( ) terminates, ob is destroyed, and its destructor is called. This causes ob.p to be freed. However, the memory freed by ob.p is the same memory that is still in use by a.p! This is, in itself, a serious bug.

However, things get even worse. When the program ends, a is destroyed, and its dynamically allocated memory is freed a second time. The problem is that freeing the same piece of dynamically allocated memory a second time is an undefined operation which could, depending upon how the dynamic allocation system is implemented, cause a fatal error.As you might guess, one way around the problem of a parameter's destructor destroying data needed by the calling argument is to pass either a pointer or a reference, insteadof the object itself. When either a pointer to an object or a reference to an object is passed, no copy is made; thus, no destructor is called when the function returns. For example, here is one way to correct the

preceding program:

```
   // One solution to the problem of passing
 objects.
   #include <iostream.h>
   #include <stdlib.h>
   class myclass
   {
        int *p;
     public:
        myclass(int i);
        ~myclass();
        intgetval() { return *p; }
   };
   myclass::myclass(int i)
   {
     cout<< "Allocating p\n";
     p = new int;
     *p = i;
   }
   myclass::~myclass()
   {
   cout<< "Freeing p\n";
   delete p;
   }
   /* This will NOT cause a problem.Because ob is
   now passed by reference, nocopy of the calling
   argument is made and thus, no object goes out-
   of-scope when display() terminates.*/
   void display(myclass&ob)
   {
       cout<<ob.getval() << '\n';
   }
   int main()
   {
     myclass a(10);
     display(a);
     return 0;
   }
```

The output from this version of the program is shown here.

```
Allocating p
10
Freeing p
```

As you can see, only one call to the destructor occurs. This is because no copy of ais made when it is passed by reference to display().Passing an object by reference is an excellent approach when the situation allows it, but it 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. Before discussing the copy constructor, let’s look at another, related situation that can also benefit from a copy constructor.