Constructors

The ENGLOBJ example shows two ways that member functions can be used to give values to the data items in an object. Sometimes, however, it's convenient if an object can initialize itself when it's first created, without requiring a separate call to a member function. Automatic initialization is carried out using a special member function called a constructor. A constructor is a member function that is executed automatically whenever an object is created.

A Counter Example

As an example, we'll create a class of objects that might be useful as a general-purpose programming element. A counter is a variable that counts things. Maybe it counts file accesses, or the number of times the user presses the Enter key, or the number of customers entering a bank. Each time such an event takes place, the counter is incremented (1 is added to it). The counter can also be accessed to find the current count.
Let's assume that this counter is important in the program and must be accessed by many different functions. In procedural languages such as C, a counter would probably be implemented as a global variable. However, as we noted in Chapter 1, global variables complicate the program's design and may be modified accidentally. This example, COUNTER, provides a counter variable that can be modified only through its member functions.

```
// object represents a counter variable
#include <iostream>
class Counter
{
private:
        unsigned int count; //count
public:
         Counter() : count(0) //constructor
                { /*empty body*/ }
        void inc_count() //increment count
                { count++; }
         int get_count() //return count
                 { return count; }
};
```

```
//////////////////////////////////////////////////////////////
int main()
{
Counter c1, c2; //define and initialize
cout << "\nc1=" << c1.get_count(); //display
cout << "\nc2=" << c2.get_count();
c1.inc_count(); //increment c1
c2.inc_count(); //increment c2
c2.inc_count(); //increment c2
cout << "\nc1=" << c1.get_count(); //display again
cout << "\nc2=" << c2.get_count();
cout << endl;
return 0;
}
```

The Counter class has one data member: count, of type unsigned int (since the count is always positive). It has three member functions: the constructor Counter(), which we'll look at in a moment; inc_count(), which adds 1 to count; and get_count(), which returns the current value of count.

Automatic Initialization

When an object of type Counter is first created, we want its count to be initialized to 0. After all, most counts start at 0. We could provide a set_count() function to do this and call it with an argument of 0, or we could provide a zero_count() function, which would always set count to 0. However, such functions would need to be executed every time we created a Counter object.

Counter c1	//every time we do this,
c1.zero_count()	//we must do this too

This is mistake prone, because the programmer may forget to initialize the object after

creating it. It's more reliable and convenient, especially when there are a great many objects of a given class, to cause each object to initialize itself when it's created. In the Counter class, the constructor Counter() does this. This function is called automatically whenever a new object of type Counter is created. Thus in main() the statement

Counter c1, c2;

creates two objects of type Counter. As each is created, its constructor, Counter (), is executed. This function sets the count variable to 0. So the effect of this single statement is to not only create two objects, but also to initialize their count variables to 0.

Same Name as the Class

There are some unusual aspects of constructor functions.

First, it is no accident that they have exactly the same name (Counter in this example) as the class of which they are members. This is one way the compiler knows they are constructors.

Second, no return type is used for constructors. Why not? Since the constructor is called automatically by the system, there's no program for it to return anything to; a return value wouldn't make sense. This is the second way the compiler knows they are constructors.

Initializer List

one of the most common tasks a constructor carries out is initializing data members. In the Counter class the constructor must initialize the count member to 0. You might think that this would be done in the constructor's function body, like this:

```
count()
{ count = 0; }
```

However, this is not the preferred approach (although it does work). Here's how you should initialize a data member:

```
count() : count(0)
{ }
```

The initialization takes place following the member function declarator but before the function body. It's preceded by a colon. The value is placed in parentheses following the

member data.

If multiple members must be initialized, they're separated by commas. The result is the initializer list (sometimes called by other names, such as the member-initialization list).

someClass() : m1(7), m2(33), m2(4) ← initializer list

{ }

Why not initialize members in the body of the constructor? The reasons are complex, but have to do with the fact that members initialized in the initializer list are given a value before the constructor even starts to execute. This is important in some situations. For example, the initializer list is the only way to initialize const member data and references. Actions more complicated than simple initialization must be carried out in the constructor body, as with ordinary functions.

Counter Output

The main() part of this program exercises the Counter class by creating two counters, c1 and c2.

It causes the counters to display their initial values, which—as arranged by the constructor—are 0. It then increments c1 once and c2 twice, and again cause the counters to display themselves. Here's the output:

```
c1=0
c2=0
c1=1
c2=2
```

If this isn't enough proof that the constructor is operating as advertised, we can rewrite the constructor to print a message when it executes.

Counter() : count(0)

{ cout << "I'm the constructor\n"; }

Now the program's output looks like this:

```
I'm the constructor
I'm the constructor
c1=0
c2=0
c1=1
c2=2
```

As you can see, the constructor is executed twice—once for c1 and once for c2—when the statement Counter c1, c2; is executed in main().

Destructors

we've seen that a special member functions—the constructor—is called automatically when an object is first created. You might guess that another function is called automatically when an object is destroyed. This is indeed the case. Such a function is called a destructor. A destructor has the same name as the constructor (which is the same as the class name) but is preceded by a tilde:

```
class Foo
{
private:
int data;
public:
Foo() : data(0) //constructor (same name as class)
{ }
~Foo() //destructor (same name with tilde)
{ }
};
```

Like constructors, destructors do not have a return value. They also take no arguments (the assumption being that there's only one way to destroy an object).

The most common use of destructors is to deallocate memory that was allocated for the object by the constructor.

Objects as Function Arguments

our next program adds some embellishments to the ENGLOBJ example. It also demonstrates some new aspects of classes:

1- Constructor overloading.
2- defining member functions outside the class
3- Defining objects as function arguments.

```
#include <iostream>
class Distance //English Distance class
{
private:
    int feet;
    float inches;
public:
     //constructor (no args)
     Distance() : feet(0), inches(0.0)
          { }
      //constructor (two args)
      Distance(int ft, float in) : feet(ft), inches(in)
          { }
     void getdist() //get length from user
          {
           cout << "\nEnter feet: "; cin >> feet;
           cout << "Enter inches: "; cin >> inches;
          }
     void showdist() //display distance
        {
          cout << feet << "\'-" << inches << '\"';
```