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 << '\"';
         }
       void add_dist( Distance, Distance ); //declaration
   };
   //add lengths d2 and d3
     void Distance::add_dist(Distance d2, Distance d3)
         {
             inches = d2.inches + d3.inches; //add the inches
             feet = 0; //(for possible carry)
             if(inches >= 12.0) //if total exceeds 12.0,
                 { //then decrease inches
                     inches -= 12.0; //by 12.0 and
                     feet++; //increase feet
                  } //by 1
              feet += d2.feet + d3.feet; //add the feet
           }
int main()
{
     Distance dist1, dist3; //define two lengths
     Distance dist2(11, 6.25); //define and initialize dist2
     dist1.getdist(); //get dist1 from user
     dist3.add_dist(dist1, dist2); //dist3 = dist1 + dist2
     //display all lengths
     cout << "\ndist1 = "; dist1.showdist();
     cout << "\ndist2 = "; dist2.showdist();
     cout << "\ndist3 = "; dist3.showdist();
     cout << endl;
return 0;
}
```

This program starts with a distance dist2 set to an initial value and adds to it a distance dist1, whose value is supplied by the user, to obtain the sum of the distances. It then displays all three distances:

```
Enter feet: 17
Enter inches: 5.75
dist1 = 17’-5.75”
dist2 = 11’-6.25”
dist3 = 29’-0”
```

## 1-Overloaded Constructors

it’s convenient to be able to give variables of type Distance a value when they are first created. That is, we would like to use definitions like
Distance width (5, 6.25); which defines an object, width, and simultaneously initializes it to a value of 5 for feet and 6.25 for inches.
To do this we write a constructor like this:

```
Distance(int ft, float in) : feet(ft), inches(in)
{ }
```

This sets the member data feet and inches to whatever values are passed as arguments to the constructor.
However, we also want to define variables of type Distance without initializing them, as we did in ENGLOBJ.

```
Distance dist1, dist2
```

In that program there was no constructor, but our definitions worked just fine. How could they work without a constructor? Because an implicit no-argument constructor is built into the program automatically by the compiler, and it’s this constructor that created the objects, even though we didn’t define it in the class. This no-argument constructor is called the default constructor. If it weren’t created automatically by the constructor, you wouldn’t be able to create objects of a class for which no constructor was defined.

Often we want to initialize data members in the default (no-argument) constructor as well. If we let the default constructor do it, we don't really know what values the data members may be given. If we care what values they may be given, we need to explicitly define the constructor. In ENGLECON we show how this looks:

```
Distance() : feet(0), inches(0.0) //default constructor
{ } //no function body, doesn't do anything
```

The data members are initialized to constant values, in this case the integer value 0 and the float value 0.0, for feet and inches respectively. Now we can use objects initialized with the no-argument constructor and be confident that they represent no distance (0 feet plus 0.0 inches) rather than some arbitrary value.

<u>Since there are now two explicit constructors with the same name, Distance(), we say the constructor is overloaded. Which of the two constructors is executed when an object is created depends on how many arguments are used in the definition:</u>

```
Distance length; // calls first constructor
Distance width(11, 6.0); // calls second constructor
```

## 2-Member Functions Defined Outside the Class

So far we've seen member functions that were defined inside the class definition. This need not always is the case. ENGLCON shows a member function, add_dist(), that is not defined within the Distance class definition. It is only declared inside the class, with the statement

```
void add_dist( Distance, Distance )
```

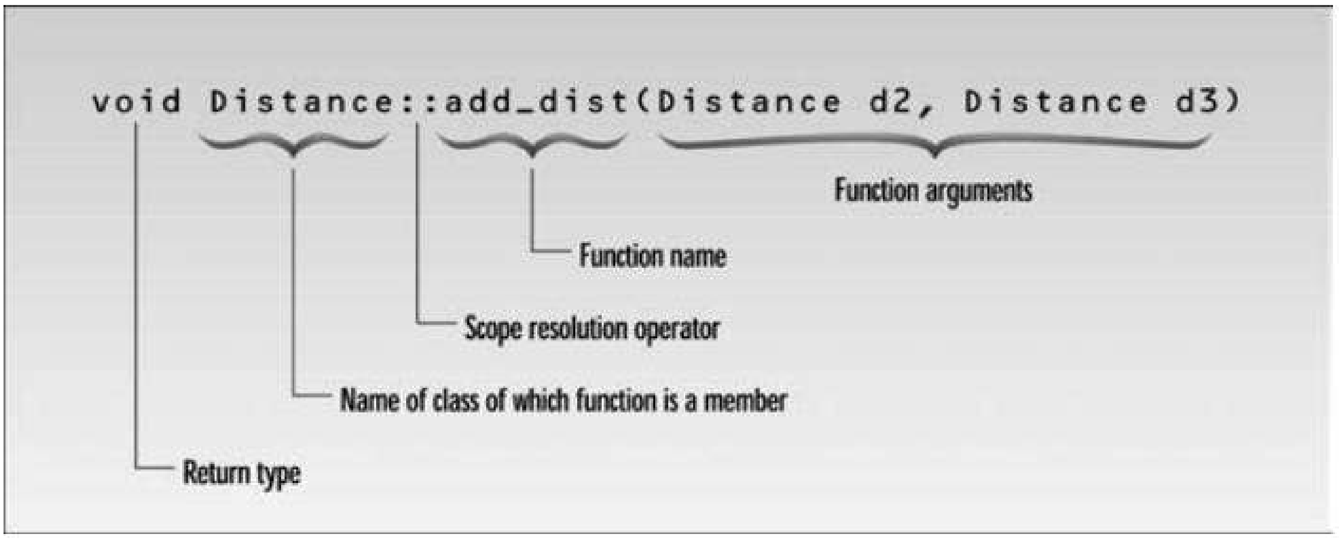
This tells the compiler that this function is a member of the class but that it will be defined outside the class declaration, someplace else in the listing.

In ENGLCON the add_dist() function is defined following the class definition.

```
//add lengths d2 and d3
void Distance::add_dist(Distance d2, Distance d3)
{
```

```
inches = d2.inches + d3.inches; //add the inches
feet = 0; //(for possible carry)
if(inches >= 12.0) //if total exceeds 12.0,
{ //then decrease inches
inches -= 12.0; //by 12.0 and
feet++; //increase feet
} //by 1
feet += d2.feet + d3.feet; //add the feet
}
```

The declarator in this definition contains some unfamiliar syntax. The function name, add_dist(), is preceded by the class name, Distance, and a new symbol—the double colon (::). This symbol is called the scope resolution operator. It is a way of specifying what class something is associated with. In this situation, Distance::add_dist() means "the add_dist() member function of the Distance class." Figure 6.5 shows its usage


void Distance::add_dist(Distance d2, Distance d3)
Function arguments
Function name
Scope resolution operator
Name of class of which function is a member
Return type

## 3-Objects as Arguments

the distances dist1 and dist3 are created using the default constructor (the one that takes no arguments). The distance dist2 is created with the constructor that takes two arguments, and is initialized to the values passed in these arguments. A value is obtained for dist1 by calling the member function getdist(), which obtains values from the user.

Now we want to add dist1 and dist2 to obtain dist3. The function call in main()

dist3.add_dist(dist1, dist2);

does this. The two distances to be added, dist1 and dist2, are supplied as arguments to add_dist(). The syntax for arguments that are objects is the same as that for arguments that are simple data types such as int: The object name is supplied as the argument. Since add_dist() is a member function of the Distance class, it can access the private data in any object of class Distance supplied to it as an argument, using names like dist1.inches and dist2.feet.

Close examination of add_dist() emphasizes some important truths about member functions.

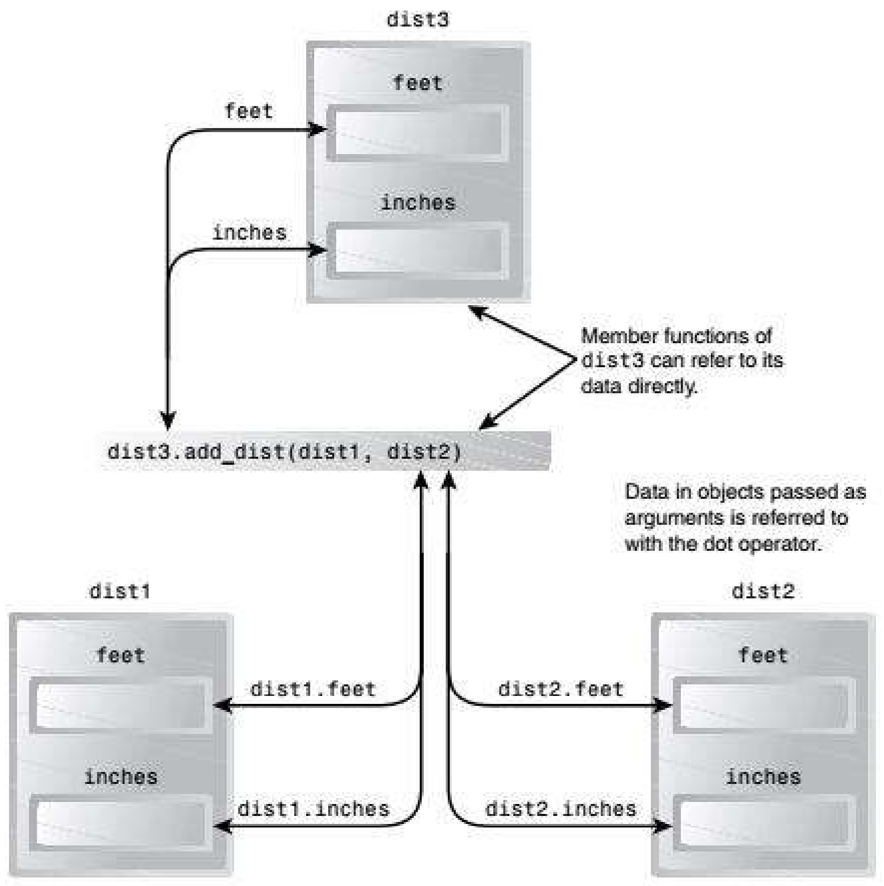
A member function is always given access to the object for which it was called: the object connected to it with the dot operator. But it may be able to access other objects. In the following statement in ENGLCON, what objects can add_dist() access?

```
dist3.add_dist(dist1, dist2)
```

Besides dist3, the object for which it was called, it can also access dist1 and dist2, because they are supplied as arguments. You might think of dist3 as a sort of phantom argument; the member function always has access to it, even though it is not supplied as an argument. That's what this statement means: "Execute the add_dist() member function of dist3." When the variables feet and inches are referred to within this function, they refer to dist3.feet and dist3.inches.

Notice that the result is not returned by the function. The return type of add_dist() is void. The result is stored automatically in the dist3 object. Figure 6.6 shows the two distances dist1 and dist2 being added together, with the result stored in dist3.

To summarize, every call to a member function is associated with a particular object (unless it's a static function; we'll get to that later). Using the member names alone (feet and inches), the function has direct access to all the members, whether private or public, of that object. It also has indirect access, using the object name and the member name, connected with the dot operator (dist1.inches or dist2.feet) to other objects of the same class that are passed as arguments.


dist3
feet
inches
feet
inches
Member functions of dist3 can refer to its data directly.
dist3.add_dist(dist1, dist2)
Data in objects passed as arguments is referred to with the dot operator.
dist1
feet
inches
dist1.feet
dist1.inches
dist2
feet
inches
dist2.feet
dist2.inches