1.5 Objects as Function Arguments

Our next program adds some new aspects of classes: constructor overloading, defining member functions outside the class, and—perhaps most importantly—objects as function arguments. Here's the listing for ENGLCON:

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

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"
```

1.6 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. So far so good. 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. 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 { }
```

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.

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:

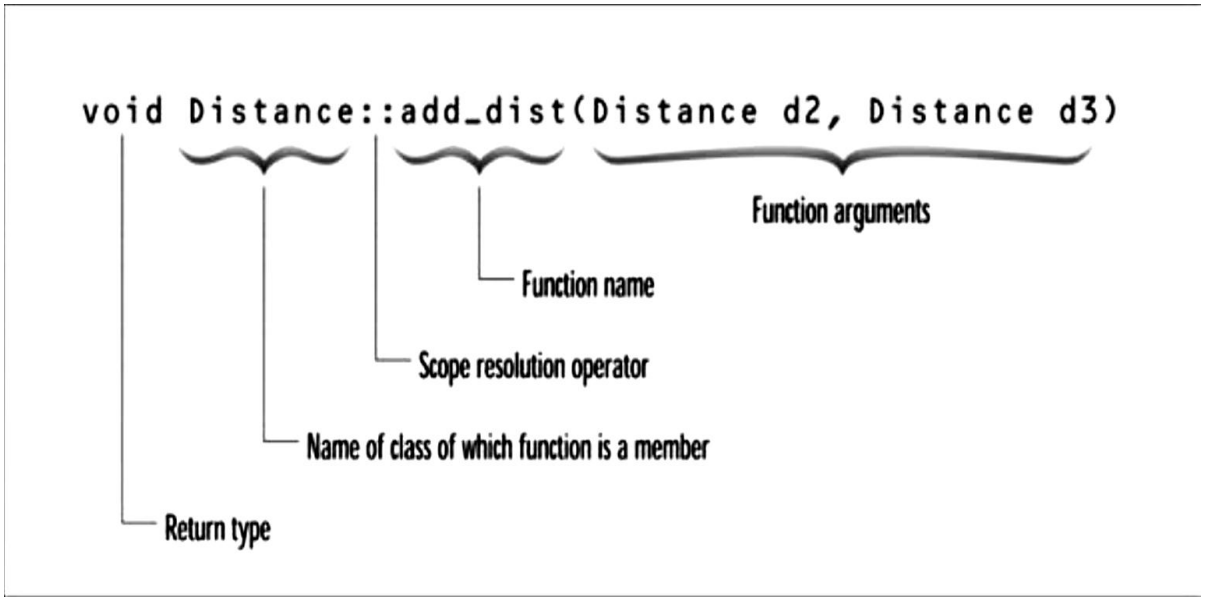
```
Distance length;          // calls first constructor
Distance width(11, 6.0); // calls second constructor
```

1.7 Member Functions Defined Outside the Class

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 something is associated with.



The scope resolution operator

1.8 Objects as Arguments

Now we can see how ENGLCON works. 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.

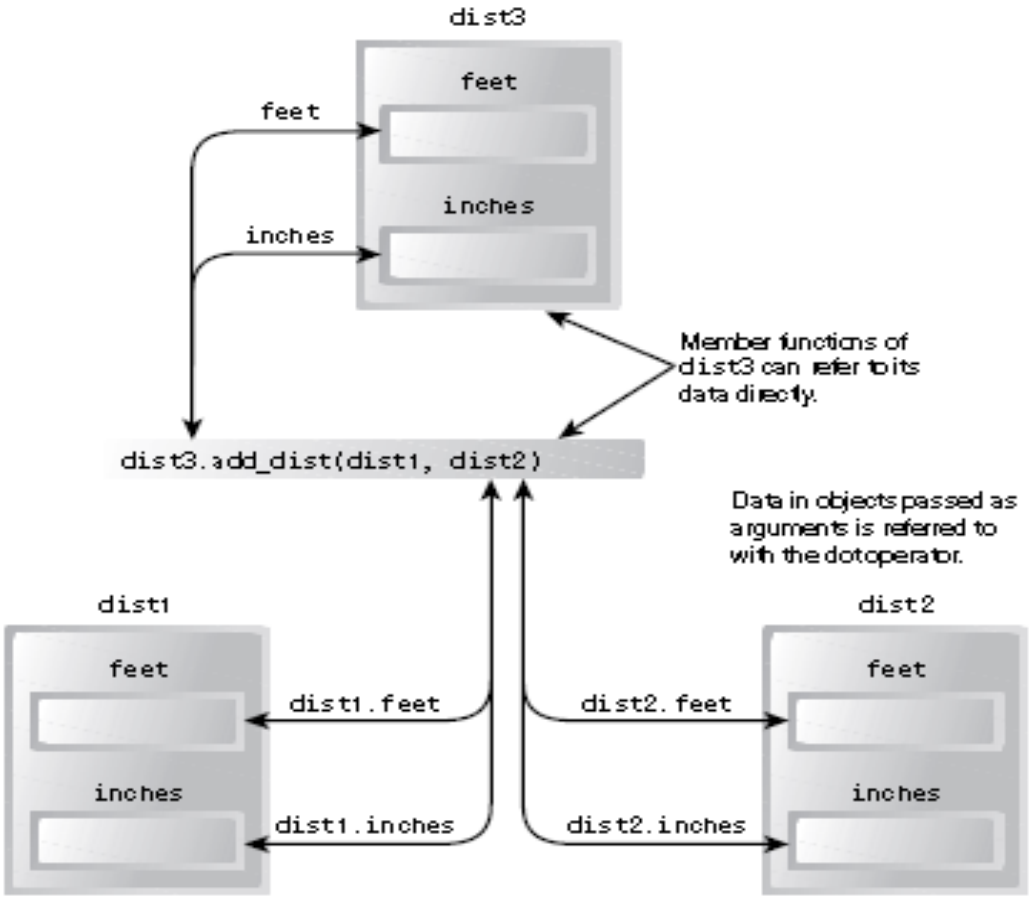
Now we want to add dist1 and dist2 to obtain dist3. The function call in main() dist3.add_dist(dist1, dist2); does this. The two distances to be added, dist1 and dist2, are supplied as arguments to add_dist().

The 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. 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 bellow shows the two distances dist1 and dist2 being added together, with the result stored in dist3.



1.9 The Default Copy Constructor

We've seen two ways to initialize objects. A no-argument constructor can initialize data members to constant values, and a multi-argument constructor can initialize data members to values passed as arguments. Let's mention another way to initialize an object: you can initialize it with another object of the same type. Surprisingly, you don't need to create a special constructor for this; one is already built into all classes. It's called the default copy constructor. It's a one argument constructor whose argument is an object of the same class as the constructor. The ECOPYCON program shows how this constructor is used.

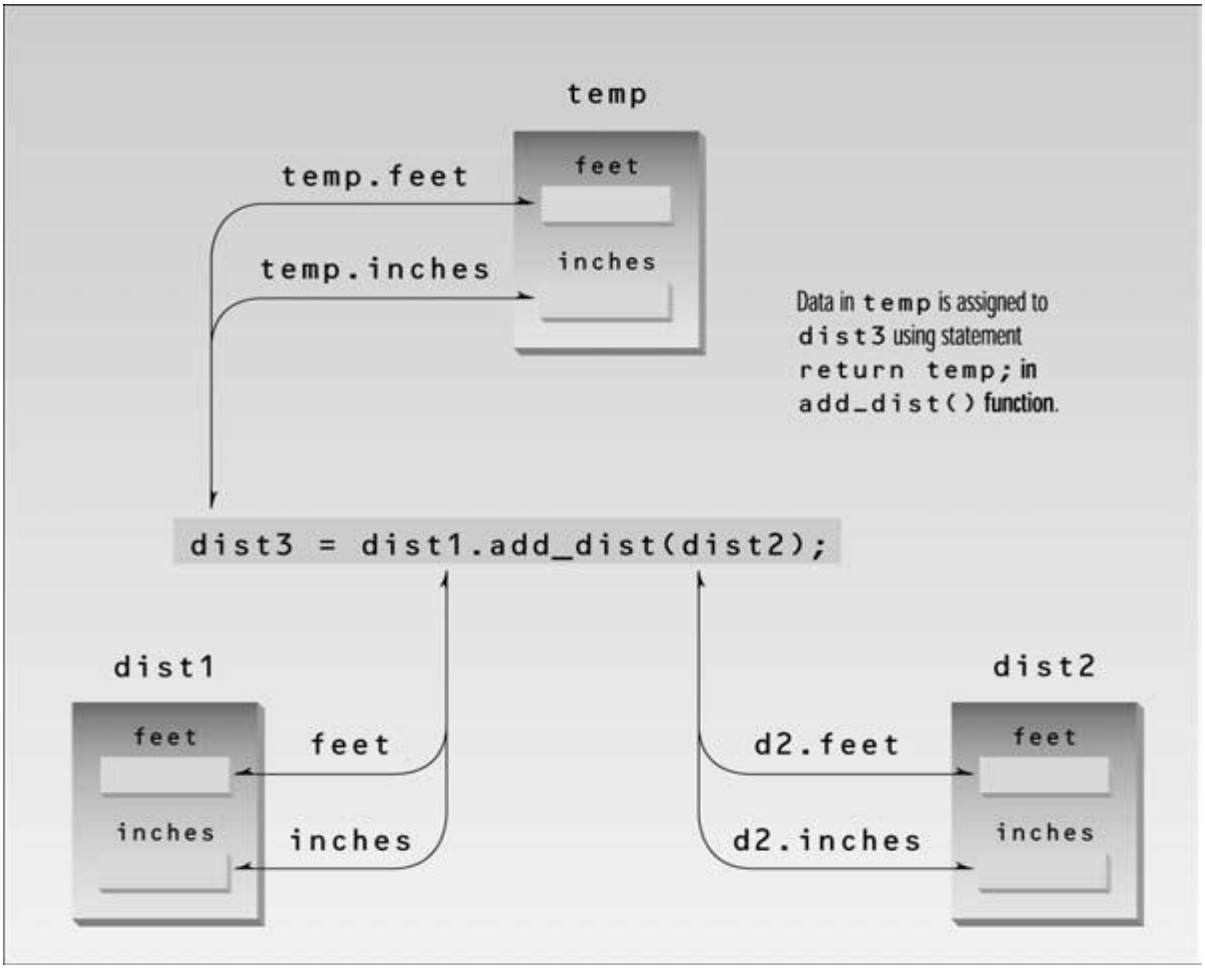
```
// ecopycon.cpp
// initialize objects using default copy constructor
#include <iostream.h>
class Distance //English Distance class
{
   private:
         int feet;
         float inches;
public:
        //constructor (no args)
       Distance() : feet(0), inches(0.0)
       { }
      //Note: no one-arg constructor
     //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 << '\"'; }
};
int main()
{
  Distance dist1(11, 6.25); //two-arg constructor
  Distance dist2(dist1); //one-arg constructor
  Distance dist3 = dist1; //also one-arg constructor
  //display all lengths
  cout << "\ndist1 = "; dist1.showdist();
  cout << "\ndist2 = "; dist2.showdist();
  cout << "\ndist3 = "; dist3.showdist();
  cout << endl;
  return 0;
}
```

1.9.1 Returning Objects from Functions

In the ENGLCON example, we saw objects being passed as arguments to functions. Now we'll see an example of a function that returns an object. We'll modify the ENGLCON program to produce ENGLRET:

```
// englret.cpp
// function returns value of type Distance
    #include <iostream.h>
    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 << '\"'; } //Distance add_dist(Distance);
            //add this distance to d2, return the sum
            Distance Distance::add_dist(Distance d2)
            {  Distance temp; //temporary variable
             temp.inches = inches + d2.inches; //add the inches
             if (temp.inches >= 12.0) //if total exceeds 12.0,
             { temp.inches -= 12.0;
                temp.feet = 1; }
              temp.feet += feet + d2.feet; //add the feet
              return temp;  }
    };
    int main()
    {          Distance dist1, dist3; //define two lengths
                Distance dist2(11, 6.25); //define, initialize dist2
                dist1.getdist(); //get dist1 from user
                dist3 = dist1.add_dist(dist2); //dist3 = dist1 + dist2 //display all lengths
                cout << "\ndist1 = "; dist1.showdist();
                cout << "\ndist2 = "; dist2.showdist();
                cout << "\ndist3 = "; dist3.showdist();
                cout << endl;
                return 0;
    }
```

The result isstored in temp and accessed as temp.feet and temp.inches. The temp object is then returned by the function using the statement return temp; and the statement in main() assigns it to dist3. Notice that dist1 is not modified; it simply supplies data to add_dist(). Figure bellow shows how this looks.



Result returned from the temporary object