Derived Class Constructors

In some languages the base class is called the superclass and the derived class is called the subclass. Some writers also refer to the base class as the parent and the derived class as the child.

There's a potential glitch in the COUNTEN program. What happens if we want to initialize a CountDn object to a value? Can the one-argument constructor in Counter be used? The answer is no. As we saw in COUNTEN, the compiler will substitute a no-argument constructor from the base class, but it draws the line at more complex constructors. To make such a definition work we must write a new set of constructors for the derived class. This is shown in the COUNTEN2 program.

```
// counten2.cpp
// constructors in derived class
#include <iostream>
class Counter
{
protected: //NOTE: not private
unsigned int count; //count
public:
Counter() : count() //constructor, no args
{ }
Counter(int c) : count(c) //constructor, one arg
{ }
unsigned int get_count() const //return count
{ return count; }
Counter operator ++ () //incr count (prefix)
```

```
{ return Counter(++count); }
};
class CountDn : public Counter
{
public:
CountDn() : Counter() //constructor, no args
{ }
CountDn(int c) : Counter(c) //constructor, 1 arg
{ }
CountDn operator -- () //decr count (prefix)
{ return CountDn(--count); }
};
////////////////////////////////////////////////////////////
int main()
{
CountDn c1; //class CountDn
CountDn c2(100);
cout << "\nc1=" << c1.get_count(); //display
cout << "\nc2=" << c2.get_count(); //display
++c1; ++c1; ++c1; //increment c1
cout << "\nc1=" << c1.get_count(); //display it
--c2; --c2; //decrement c2
cout << "\nc2=" << c2.get_count(); //display it
CountDn c3 = --c2; //create c3 from c2
cout << "\nc3=" << c3.get_count(); //display c3
cout << endl;
return 0;
}
```

This program uses two new constructors in the CountDn class. Here is the no-argument constructor:

CountDn() : Counter()

{ }

This constructor has an unfamiliar feature: the function name following the colon. This construction causes the CountDn() constructor to call the Counter() constructor in the base class. In main(), when we say CountDn c1;

the compiler will create an object of type CountDn and then call the CountDn constructor to initialize it. This constructor will in turn call the Counter constructor, which carries out the work. The CountDn() constructor could add additional statements of its own, but in this case it doesn't need to, so the function body between the braces is empty. Calling a constructor from the initialization list may seem odd, but it makes sense. You want to initialize any variables, whether they're in the derived class or the base class, before any statements in either the derived or base-class constructors are executed. By calling the base class constructor before the derived-class constructor starts to execute, we accomplish this.

The statement CountDn c2(100);

in main() uses the one-argument constructor in CountDn. This constructor also calls the corresponding one-argument constructor in the base class: CountDn(int c) : Counter(c) ←□□□ argument c is passed to Counter { }

This construction causes the argument c to be passed from CountDn() to Counter(), where it is used to initialize the object.

In main(), after initializing the c1 and c2 objects, we increment one and decrement the other and then print the results. The one-argument constructor is also used in an assignment statement. CountDn c3 = --c2;

Inheritance in the English Distance Class

Here's a somewhat more complex example of inheritance. So far in this book the various programs that used the English Distance class assumed that the distances to be represented would always be positive. This is usually the case in architectural drawings. However, if we were measuring, say, the water level of the Pacific Ocean as the tides varied, we might want to be able to represent negative feet-and-inches quantities. (Tide levels below mean-lower-low-water are called minus tides; they prompt clam diggers to take advantage of the larger area of exposed beach.)

Let's derive a new class from Distance. This class will add a single data item to our feet-and inches measurements: a sign, which can be positive or negative. When we add the sign, we'll also need to modify the member functions so they can work with signed distances. Here's the listing for ENGLEN:

```
// englen.cpp
// inheritance using English Distances
#include <iostream>
using namespace std;
enum posneg { pos, neg }; //for sign in DistSign
class Distance //English Distance class
{
protected: //NOTE: can't be private
```

```
int feet; float inches;
public: //no-arg constructor
Distance() : feet(0), inches(0.0) { } //2-arg constructor)
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() const //display distance
{ cout << feet << "\'-" << inches << '\"'; }
};
class DistSign : public Distance //adds sign to Distance
{
private:
posneg sign; //sign is pos or neg
public:
DistSign() : Distance() //call base constructor no-arg constructor
{ sign = pos; } //set the sign to + Inheritance
//2- or 3-arg constructor
DistSign(int ft, float in, posneg sg=pos) : Distance(ft, in) //call base constructor
{ sign = sg; } //set the sign
void getdist() //get length from user
{ Distance::getdist(); //call base getdist()
char ch; //get sign from user
cout << "Enter sign (+ or -): "; cin >> ch;
sign = (ch=='+') ? pos : neg; }
void showdist() const //display distance
{ cout << ( (sign==pos) ? "(+)" : "(-)" ); //show sign
Distance::showdist(); //ft and in }
```

```
};
int main()
{
DistSign alpha; //no-arg constructor
alpha.getdist(); //get alpha from user
DistSign beta(11, 6.25); //2-arg constructor
DistSign gamma(100, 5.5, neg); //3-arg constructor
cout << "\nalpha = "; alpha.showdist();//display all distances
cout << "\nbeta = "; beta.showdist();
cout << "\ngamma = "; gamma.showdist();
cout << endl;
return 0;
}
```

Here the DistSign class adds the functionality to deal with signed numbers. The Distance class in this program is just the same as in previous programs, except that the data is protected.

Actually in this case it could be private, because none of the derived-class functions accesses it. However, it's safer to make it protected so that a derived-class function could access it if necessary.

The main() program declares three different signed distances. It gets a value for alpha from the user and initializes beta to (+)11'–6.25" and gamma to (–)100'–5.5". In the output we use parentheses around the sign to avoid confusion with the hyphen separating feet and inches. Here's some sample output:

```
Enter feet: 6
Enter inches: 2.5
```

```
Enter sign (+ or -): -
alpha = (-)6'-2.5"
beta = (+)11'-6.25"
gamma = (-)100'-5.5"
```

The DistSign class is derived from Distance. It adds a single variable, sign, which is of type posneg. The sign variable will hold the sign of the distance. The posneg type is defined in an enum statement to have two possible values: pos and neg.

Constructors in DistSign

DistSign has two constructors, mirroring those in Distance. The first takes no arguments, the second takes either two or three arguments. The third, optional, argument in the second constructor is a sign, either pos or neg. Its default value is pos. These constructors allow us to define variables (objects) of type DistSign in several ways.

Both constructors in DistSign call the corresponding constructors in Distance to set the feetand-inches values. They then set the sign variable. The no-argument constructor always sets it to pos. The second constructor sets it to pos if no third-argument value has been provided, or to a value (pos or neg) if the argument is specified.

The arguments ft and in, passed from main() to the second constructor in DistSign, are simply forwarded to the constructor in Distance. Member Functions in DistSign Adding a sign to Distance has consequences for both of its member functions. The getdist() function in the DistSign class must ask the user for the sign as well as for feet-and-inches values, and the showdist() function must display the sign

along with the feet and inches. These functions call the corresponding functions in Distance, in the lines

Distance::getdist(); and Distance::showdist();

These calls get and display the feet and inches values. The body of getdist() and showdist() in DistSign then go on to deal with the sign.