## 7.5 Overriding Member Functions

**You can use member functions in a derived class that override—that is, have the same name as—those in the base class. You might want to do this so that calls in your program work the same way for objects of both base and derived classes. The program modeled a stack, a simple data storage device. It allowed you to push integers onto the stack and pop them off. However, STAKARAY had a potential flaw. If you tried to push too many items onto the stack, the program might bomb, since data would be placed in memory beyond the end of the st[ ] array. Or if you tried to pop too many items, the results would be meaningless, since you would be reading data from memory locations outside the array. To cure these defects we've created a new class, Stack2, derived from Stack. Objects of Stack2 behave in exactly the same way as those of Stack, except that you will be warned if you attempt to push too many items on the stack or if you try to pop an item from an empty stack. Here's the listing for STAKEN:**

```
// staken.cpp
#include <iostream.h>
#include <process.h> //for exit()
class Stack
{
protected: //NOTE: can't be private
enum { MAX = 3 }; //size of stack array
int st[MAX]; //stack: array of integers
int top; //index to top of stack
public:
Stack() //constructor
{ top = -1; }
void push(int var) //put number on stack
{ st[++top] = var; }
int pop() //take number off stack
{ return st[top--]; }
};
class Stack2 : public Stack
{
public:
void push(int var) //put number on stack
{
if(top >= MAX-1) //error if stack full
{ cout << "\nError: stack is full"; exit(1); }
Stack::push(var); //call push() in Stack class
}
int pop() //take number off stack
{
if(top < 0) //error if stack empty
{ cout << "\nError: stack is empty\n"; exit(1); }
return Stack::pop(); //call pop() in Stack class
}
};
int main()
{
Stack2 s1;
s1.push(11); //push some values onto stack
```

```
s1.push(22);
s1.push(33);
cout << endl << s1.pop(); //pop some values from stack
cout << endl << s1.pop();
cout << endl << s1.pop();
cout << endl << s1.pop(); //oops, popped one too many...
cout << endl;
return 0;
}
```

**In this program the Stack class is just the same as it was in the STAKARAY program, except that the data members have been made protected.**

## 7.6 Which Function Is Used?

**The Stack2 class contains two functions, push() and pop(). These functions have the same names, and the same argument and return types, as the functions in Stack. When we call these functions from main(), in statements like**

```
s1.push(11);
```

**how does the compiler know which of the two push() functions to use? Here's the rule: When the same function exists in both the base class and the derived class, the function in the derived class will be executed. (This is true of objects of the derived class. Objects of the base class don't know anything about the derived class and will always use the base class functions.) We say that the derived class function overrides the base class function. So in the preceding statement, since s1 is an object of class Stack2, the push() function in Stack2 will be executed, not the one in Stack. The push() function in Stack2 checks to see whether the stack is full. If it is, it displays an error message and causes the program to exit. If it isn't, it calls the push() function in Stack. Similarly, the pop() function in Stack2 checks to see whether the stack is empty. If it is, it prints an error message and exits; otherwise, it calls the pop() function in Stack. In main() we push three items onto the stack, but we pop four. The last pop elicits an error message**

```
33
22
11
Error: stack is empty
```

**and terminates the program.**

### 7.6.1 Scope Resolution with Overridden Functions

**How do push() and pop() in Stack2 access push() and pop() in Stack? They use the scope resolution operator, ::, in the statements**

```
Stack::push(var);
```

**and**

```
return Stack::pop();
```

**These statements specify that the push() and pop() functions in Stack are to be called. Without the scope resolution operator, the compiler would think the push() and pop() functions in Stack2 were calling themselves, which—in this case—would lead to program failure. Using the scope resolution operator allows you to specify exactly what class the function is a member of.**

## 7.7 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.h>
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:
//no-arg constructor
DistSign() : Distance() //call base constructor
{ sign = pos; } //set the sign to +
//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
//display all distances
cout << "\nalpha = "; alpha.showdist();
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.**

## 7.7.1 Operation of ENGLEN

**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.**

## 7.7.2 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.**

## 7.7.3 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.**

## Exercises

**1. Imagine a publishing company that markets both book and audiocassette versions of its works. Create a class publication that stores the title (a int) and price (type float) of a publication. From this class derive two classes: book, which adds a page count (type int), and tape, which adds a playing time in minutes (type float). Each of these three classes should have a getdata() function to get its data from the user at the keyboard, and a putdata() function to display its data. Write a main() program to test the book and tape classes by creating instances of them, asking the user to fill in data with getdata(), and then displaying the data with putdata().**

**2. Start with the publication, book, and tape classes of Exercise 1. Add a base class sales that holds an array of three floats so that it can record the dollar sales of a particular publication for the last three months. Include a getdata() function to get three sales amounts from the user, and a putdata() function to display the sales figures. Alter the book and tape classes so they are derived from both publication and sales. An object of class book or tape should input and output sales data along with its other data. Write a main() function to create a book object and a tape object and exercise their input/output capabilities.**

**Solutions to Exercises**

**1.**

```
// ex1.cpp
// publication class and derived classes
#include <iostream.h>
#include <string.h>
class publication // base class
{
private:
int title;
float price;
public:
```

```
void getdata()
{
cout << "\nEnter title: "; cin >> title;
cout << "Enter price: "; cin >> price;
}
void putdata() const
{
cout << "\nTitle: " << title;
cout << "\nPrice: " << price;
}
};
////////////////////////////////////////////////////////////
class book : private publication // derived class
{
private:
int pages;
public:
void getdata()
{
publication::getdata();
cout << "Enter number of pages: "; cin >> pages;
}
void putdata() const
{
publication::putdata();
cout << "\nPages: " << pages;
}
};
////////////////////////////////////////////////////////////
class tape : private publication // derived class
{
private:
float time;
public:
void getdata()
{
publication::getdata();
cout << "Enter playing time: "; cin >> time;
}
void putdata() const
{
publication::putdata();
cout << "\nPlaying time: " << time;
}
};
////////////////////////////////////////////////////////////
int main()
{
book book1; // define publications
tape tape1;
```

```
book1.getdata(); // get data for them
tape1.getdata();
book1.putdata(); // display their data
tape1.putdata();
cout << endl;
return 0;
}
```

**2.**

```
// ex2.cpp
// multiple inheritance with publication class
#include <iostream.h>
#include <string.h>
////////////////////////////////////////////////////////////
class publication
{
private:
int title;
float price;
public:
void getdata()
{
cout << "\nEnter title: "; cin >> title;
cout << " Enter price: "; cin >> price;
}
void putdata() const
{
cout << "\nTitle: " << title;
cout << "\n Price: " << price;
}
};
////////////////////////////////////////////////////////////
class sales
{
private:
enum { MONTHS = 3 };
float salesArr[MONTHS];
public:
void getdata();
void putdata() const;
};
//-----------------------------------------------------------
void sales::getdata()
{
cout << " Enter sales for 3 months\n";
for(int j=0; j<MONTHS; j++)
{
cout << " Month " << j+1 << ": ";
cin >> salesArr[j];
}
```

```
}
//-----------------------------------------------------------
void sales::putdata() const
{
for(int j=0; j<MONTHS; j++)
{
cout << "\n Sales for month " << j+1 << ": ";
cout << salesArr[j];
}
}
//////////////////////////////////////////////////////////////
class book : private publication, private sales
{
private:
int pages;
public:
void getdata()
{
publication::getdata();
cout << " Enter number of pages: "; cin >> pages;
sales::getdata();
}
void putdata() const
{
publication::putdata();
cout << "\n Pages: " << pages;
sales::putdata();
}
};
//////////////////////////////////////////////////////////////
class tape : private publication, private sales
{
private:
float time;
public:
void getdata()
{
publication::getdata();
cout << " Enter playing time: "; cin >> time;
sales::getdata();
}
void putdata() const
{
publication::putdata();
cout << "\n Playing time: " << time;
sales::putdata();
}
};
//////////////////////////////////////////////////////////////
int main()
```

```
{
book book1; // define publications
tape tape1;
book1.getdata(); // get data for publications
tape1.getdata();
book1.putdata(); // display data for publications
tape1.putdata();
cout << endl;
return 0;
}
```