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. Here's an example based on the STAKARAY program. This program models a stack, a simple data storage device. It allows you to push integers onto the stack and pop them off. However, STAKARAY has 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
// overloading functions in base and derived classes
#include <iostream>
using namespace std;
#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.

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.

Scope Resolution with Overridden Functions

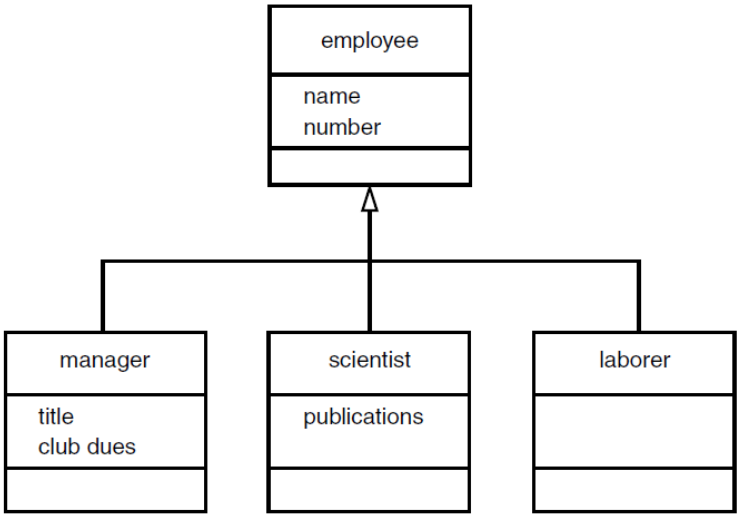
How do push() and pop() in Stack2 access push() and pop() in Stack? They use the scope resolution operator, ::, in the statements Stack::push(var); And return Stack::pop();

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.

Class Hierarchies

inheritance has been used to add functionality to an existing class. Now let's look at an example where inheritance is used for a different purpose: as part of the original design of a program. Our example models a database of employees of a widget company. We've simplified the situation so that only three kinds of employees are represented. Managers manage, scientists perform research to develop better widgets, and laborers operate the dangerous widget-stamping presses Our example program starts with a base class employee. This class handles the employee's last name and employee number. From this class three other classes are derived: manager, scientist, and laborer. The manager and scientist classes contain additional information about these categories of employee, and member functions to handle this information, as shown in Figure



Here's the listing for EMPLOY:

```
// employ.cpp
// models employee database using inheritance
#include <iostream>
using namespace std;
const int LEN = 80; //maximum length of names
////////////////////////////////////////////////////////////////
class employee //employee class
{
private:
char name[LEN]; //employee name
unsigned long number; //employee number
public:
void getdata()
{
cout << "\n Enter last name: "; cin >> name;
cout << " Enter number: "; cin >> number;

void putdata() const
```

```
{
cout << "\n Name: " << name;
cout << "\n Number: " << number;
}
};
////////////////////////////////////////////////////////////
class manager : public employee //management class
{
private:
char title[LEN]; //"vice-president" etc.
double dues; //golf club dues
public:
void getdata()
{
employee::getdata();
cout << " Enter title: "; cin >> title;
cout << " Enter golf club dues: "; cin >> dues;
}
void putdata() const
{
employee::putdata();
cout << "\n Title: " << title;
cout << "\n Golf club dues: " << dues;
}
};
////////////////////////////////////////////////////////////
class scientist : public employee //scientist class
{
private:
int pubs; //number of publications
public:
void getdata()
{
employee::getdata();
cout << " Enter number of pubs: "; cin >> pubs;
}
void putdata() const
{
employee::putdata();
cout << "\n Number of publications: " << pubs;
}
};
////////////////////////////////////////////////////////////
class laborer : public employee //laborer class

};

//////////////////////////////////////////////////////////////

int main()

{

manager m1, m2;

scientist s1;

laborer l1;

cout << endl; //get data for several employees

cout << "\nEnter data for manager 1";
```

```
m1.getdata();

cout << "\nEnter data for manager 2";

m2.getdata();

cout << "\nEnter data for scientist 1";

s1.getdata();

cout << "\nEnter data for laborer 1";

l1.getdata();

//display data for several employees

cout << "\nData on manager 1";

m1.putdata();

cout << "\nData on manager 2";

m2.putdata();

cout << "\nData on scientist 1";

s1.putdata();

cout << "\nData on laborer 1";

l1.putdata();

cout << endl;

return 0;
}
```

The main() part of the program declares four objects of different classes: two managers, a scientist, and a laborer. (Of course many more employees of each type could be defined, but the output would become rather large.) It then calls the getdata() member functions to obtain information about each employee, and the putdata() function to display this information.

Here's a sample interaction with EMPLOY. First the user supplies the data.

```
Enter data for manager 1

Enter last name: Wainsworth

Enter number: 10
```

Enter title: President

Enter golf club dues: 1000000

Enter data on manager 2

Enter last name: Bradley

Enter number: 124

Enter title: Vice-President

Enter golf club dues: 500000

Enter data for scientist 1

Enter last name: Hauptman-Frenglish

Enter number: 234234

Enter number of pubs: 999

Enter data for laborer 1

Enter last name: Jones

Enter number: 6546544

The program then plays it back.

Data on manager 1

Name: Wainsworth

Number: 10

Title: President

Golf club dues: 1000000

Data on manager 2

Name: Bradley

Number: 124

Title: Vice-President

Golf club dues: 500000

Data on scientist 1

Name: Hauptman-Frenglish

Number: 234234

Number of publications: 999

Data on laborer 1

Name: Jones

Number: 6546544

A more sophisticated program would use an array or some other container to arrange the data so that a large number of employee objects could be accommodated.

“Abstract” Base Class

Notice that we don’t define any objects of the base class employee. We use this as a general class whose sole purpose is to act as a base from which other classes are derived. The laborer class operates identically to the employee class, since it contains no additional data or functions. It may seem that the laborer class is unnecessary, but by making it a separate class we emphasize that all classes are descended from the same source, employee. Also, if in the future we decided to modify the laborer class, we would not need to change the declaration for employee Classes used only for deriving other classes, as employee is in EMPLOY, are sometimes loosely called abstract classes, meaning that no actual instances (objects) of this class are created.

However, the term abstract has a more precise definition that we’ll look at in next lecture, “Virtual Functions.”