Inheritance

Inheritance

With all appliances and means to boot.

WILLIAM SHAKESPEARE, *KING HENRY IV*, PART III

Introduction

Object-oriented programming is a popular and powerful programming technique. Among other things, it provides for a new dimension of abstraction known as *inheritance.* This means that a very general form of a class can be defined and compiled. Later, more specialized versions of that class can be defined and can inherit all the properties of the previous class. Provisions for **object-oriented programming** were a part of the initial design of C++, and facilities for inheritance are available in all versions of C++.

Prerequisites

Section 16.1 uses material from Chapters 2 to 11. Sections 16.2 and 16.3 use material from Chapters 12 and 15 in addition to Chapters 2 to 11 and Section 16.1.

16.1 Inheritance Basics

If there is anything that we wish to change in the child, we should first examine it and see whether it is not something that could better be changed in ourselves.

CARL GUSTAV JUNG, *THE INTEGRATION OF THE PERSONALITY*

inheritance
derived class
base class

One of the most powerful features of C++ is the use of inheritance to derive one class from another. **Inheritance** is the process by which a new class—known as a **derived class**—is created from another class, called the **base class.** A derived class automatically has all the member variables and functions that the base class has, and can have additional member functions and/or additional member variables.

In Chapter 5, we noted that saying that class `D` is derived from another class `B` means that class `D` has all the features of class `B` and some extra, added features as

well. When a class D is derived from a class B, we say that B is the base class and D is the derived class. We also say that D is the **child class** and B is the **parent class**.[1]

child class
parent class

For example, we discussed the fact that the predefined class ifstream is derived from the (predefined) class istream by adding member functions such as open and close. The stream cin belongs to the class of all input streams (that is, the class istream), but it does not belong to the class of input-file streams (that is, does not belong to ifstream), partly because it lacks the member functions open and close of the derived class ifstream.

Derived Classes

Suppose we are designing a record-keeping program that has records for salaried employees and hourly employees. There is a natural hierarchy for grouping these classes. These are all classes of people who share the property of being employees.

Employees who are paid an hourly wage are one subset of employees. Another subset consists of employees who are paid a fixed wage each month or each week. Although the program may not need any type corresponding to the set of all employees, thinking in terms of the more general concept of employees can be useful. For example, all employees have names and social security numbers, and the member functions for setting and changing names and social security numbers will be the same for salaried and hourly employees.

Within C++ you can define a class called Employee that includes all employees, whether salaried or hourly, and then use this class to define classes for hourly employees and salaried employees. Displays 16.1 and 16.2 show one possible definition for the class Employee.

You can have an (undifferentiated) Employee object, but our reason for defining the class Employee is so that we can define derived classes for different kinds of employees. In particular, the function print_check will always have its definition changed in derived classes so that different kinds of employees can have different kinds of checks. This is reflected in the definition of the function print_check for the class Employee (Display 16.2). It makes little sense to print a check for such an (undifferentiated) Employee. We know nothing about this employee's salary details. Consequently we implemented the function print_check of the class Employee so that the program stops with an error message if print_check is called for a base class Employee object. As you will see, derived classes will have enough information to redefine the function print_check to produce meaningful employee checks.

[1] Some authors speak of a *subclass* D and *superclass* B instead of derived class D and base class B. However we have found the terms *derived class* and *base class* to be less confusing. We only mention this in an effort to help you to read other texts.

Display 16.1 Interface for the Base Class `Employee`

```
//This is the header file employee.h.
//This is the interface for the class Employee.
//This is primarily intended to be used as a base class to derive
//classes for different kinds of employees.
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>
using namespace std;

namespace employeessavitch
{

    class Employee
    {
    public:
        Employee( );
        Employee(string the_name, string the_ssn);
        string get_name( ) const;
        string get_ssn( ) const;
        double get_net_pay( ) const;
        void set_name(string new_name);
        void set_ssn(string new_ssn);
        void set_net_pay(double new_net_pay);
        void print_check( ) const;
    private:
        string name;
        string ssn;
        double net_pay;
    };

}//employeessavitch

#endif //EMPLOYEE_H
```

Display 16.2 Implementation for the Base Class `Employee` (*part 1 of 2*)

```
//This is the file: employee.cpp.
//This is the implementation for the class Employee.
//The interface for the class Employee is in the header file employee.h.
#include <string>
#include <cstdlib>
#include <iostream>
#include "employee.h"
using namespace std;

namespace employeessavitch
{
    Employee::Employee( ) : name("No name yet"), ssn("No number yet"), net_pay(0)
    {
        //deliberately empty
    }

    Employee::Employee(string the_name, string the_number)
       : name(the_name), ssn(the_number), net_pay(0)
    {
        //deliberately empty
    }

    string Employee::get_name( ) const
    {
        return name;
    }

    string Employee::get_ssn( ) const
    {
        return ssn;
    }
```

Display 16.2 Implementation for the Base Class `Employee` (*part 2 of 2*)

```
double Employee::get_net_pay( ) const
{
    return net_pay;
}

void Employee::set_name(string new_name)
{
    name = new_name;
}

void Employee::set_ssn(string new_ssn)
{
    ssn = new_ssn;
}

void Employee::set_net_pay (double new_net_pay)
{
    net_pay = new_net_pay;
}

void Employee::print_check( ) const
{
    cout << "\nERROR: print_check FUNCTION CALLED FOR AN \n"
         << "UNDIFFERENTIATED EMPLOYEE. Aborting the program.\n"
         << "Check with the author of the program about this bug.\n";
    exit(1);
}

}//employeessavitch
```

A class that is derived from the class Employee will automatically have all the member variables of the class Employee (name, ssn, and net_pay). A class that is derived from the class Employee will also have all the member functions of the class Employee, such as print_check, get_name, set_name, and the other member functions listed in Display 16.1. This is usually expressed by saying that the derived class **inherits** the member variables and member functions.

inherits

The interface files with the class definitions of two derived classes of the class Employee are given in Displays 16.3 (HourlyEmployee) and 16.4 (SalariedEmployee). We have placed the class Employee and the two derived classes in the same namespace. C++ does not require that they be in the same namespace, but since they are related classes, it makes sense to put them in the same namespace. We will first discuss the derived class HourlyEmployee given in Display 16.3.

Note that the definition of a derived class begins like any other class definition but adds a colon, the reserved word *public*, and the name of the base class to the first line of the class definition, as in the following (from Display 16.3):

```
class HourlyEmployee : public Employee
{
```

The derived class (such as HourlyEmployee) automatically receives all the member variables and member functions of the base class (such as Employee) and can add additional member variables and member functions.

The definition of the class HourlyEmployee does not mention the member variables name, ssn, and net_pay, but every object of the class HourlyEmployee has member variables named name, ssn, and net_pay. The member variables name, ssn, and net_pay are inherited from the class Employee. The class HourlyEmployee declares two additional member variables named wage_rate and hours. Thus, every object of the class HourlyEmployee has five member variables named name, ssn, net_pay, wage_rate, and hours. Note that the definition of a derived class (such as HourlyEmployee) only lists the added member variables. The member variables defined in the base class are not mentioned. They are provided automatically to the derived class.

Just as it inherits the member variables of the class Employee, the class HourlyEmployee inherits all the member functions from the class Employee. So, the class HourlyEmployee inherits the member functions get_name, get_ssn, get_net_pay, set_name, set_ssn, set_net_pay, and print_check from the class Employee.

In addition to the inherited member variables and member functions, a derived class can add new member variables and new member functions. The new member variables and the declarations for the new member functions are listed in the class definition. For example, the derived class HourlyEmployee adds the two member variables wage_rate and hours, and it adds the new member functions named

Display 16.3 Interface for the Derived Class `HourlyEmployee`

```
//This is the header file hourlyemployee.h.
//This is the interface for the class HourlyEmployee.
#ifndef HOURLYEMPLOYEE_H
#define HOURLYEMPLOYEE_H

#include <string>
#include "employee.h"

using namespace std;

namespace employeessavitch
{

    class HourlyEmployee : public Employee
    {
    public:
        HourlyEmployee( );
        HourlyEmployee(string the_name, string the_ssn,
                       double the_wage_rate, double the_hours);
        void set_rate(double new_wage_rate);
        double get_rate( ) const;
        void set_hours(double hours_worked);
        double get_hours( ) const;
        void print_check( ) ;
    private:
        double wage_rate;
        double hours;
    };

}//employeessavitch

#endif //HOURLYMPLOYEE_H
```

You only list the declaration of an inherited member function if you want to change the definition of the function.

Display 16.4 Interface for the Derived Class `SalariedEmployee`

```
//This is the header file salariedemployee.h.
//This is the interface for the class SalariedEmployee.
#ifndef SALARIEDEMPLOYEE_H
#define SALARIEDEMPLOYEE_H

#include <string>
#include "employee.h"

using namespace std;

namespace employeessavitch
{

    class SalariedEmployee : public Employee
    {
    public:
        SalariedEmployee( );
        SalariedEmployee (string the_name, string the_ssn,
                                  double the_weekly_salary);
        double get_salary( ) const;
        void set_salary(double new_salary);
        void print_check( );
    private:
        double salary;//weekly
    };

}//employeessavitch

#endif //SALARIEDEMPLOYEE_H
```

Inherited Members

A derived class automatically has all the member variables and all the ordinary member functions of the base class. (As discussed later in this chapter, there are some specialized member functions, such as constructors, that are not automatically inherited.) These members from the base class are said to be **inherited.** These inherited member functions and inherited member variables are, with one exception, not mentioned in the definition of the derived class, but they are automatically members of the derived class. As explained in the text, you do mention an inherited member function in the definition of the derived class if you want to change the definition of the inherited member function.

`set_rate`, `get_rate`, `set_hours`, and `get_hours`. This is shown in Display 16.3. Note that you do not give the declarations of the inherited member functions except for those whose definitions you want to change, which is the reason we list only the member function `print_check` from the base class `Employee`. For now, do not worry about the details of the constructor definition for the derived class. We will discuss constructors in the next subsection.

In the implementation file for the derived class, such as the implementation of `HourlyEmployee` in Display 16.5, you give the definitions of all the added member functions. Note that you do not give definitions for the inherited member functions unless the definition of the member function is changed in the derived class, a point we discuss next.

redefining

The definition of an inherited member function can be changed in the definition of a derived class so that it has a meaning in the derived class that is different from what it is in the base class. This is called **redefining** the inherited member function. For example, the member function `print_check( )` is redefined in the definition of the derived class `HourlyEmployee`. To redefine a member function definition, simply

Parent and Child Classes

When discussing derived classes, it is common to use terminology derived from family relationships. A base class is often called a **parent class.** A derived class is then called a **child class.** This makes the language of inheritance very smooth. For example, we can say that a child class inherits member variables and member functions from its parent class. This analogy is often carried one step further. A class that is a parent of a parent of a parent of another class (or some other number of "parent of" iterations) is often called an **ancestor class.** If class A is an ancestor of class B, then class B is often called a **descendant** of class A.

Display 16.5 Implementation for the Derived Class `HourlyEmployee` (*part 1 of 2*)

```
//This is the file: hourlyemployee.cpp
//This is the implementation for the class HourlyEmployee.
//The interface for the class HourlyEmployee is in
//the header file hourlyemployee.h.
#include <string>
#include <iostream>
#include "hourlyemployee.h"
using namespace std;

namespace employeessavitch
{

    HourlyEmployee::HourlyEmployee( ) : Employee( ), wage_rate(0), hours(0)
    {
        //deliberately empty
    }

    HourlyEmployee::HourlyEmployee(string the_name, string the_number,
                                   double the_wage_rate, double the_hours)
    : Employee(the_name, the_number), wage_rate(the_wage_rate), hours(the_hours)
    {
        //deliberately empty
    }

    void HourlyEmployee::set_rate(double new_wage_rate)
    {
        wage_rate = new_wage_rate;
    }

    double HourlyEmployee::get_rate( ) const
    {
        return wage_rate;
    }
```

Display 16.5 Implementation for the Derived Class `HourlyEmployee` (*part 2 of 2*)

```
void HourlyEmployee::set_hours(double hours_worked)
{
    hours = hours_worked;
}

double HourlyEmployee::get_hours( ) const
{
    return hours;
}
```

We have chosen to set `net_pay` *as part of the* `print_check` *function since that is when it is used, but in any event, this is an accounting question, not a programming question. But note that C++ allows us to drop the* `const` *in the function* `print_check` *when we redefine it in a derived class.*

```
void HourlyEmployee::print_check( )
{
    set_net_pay(hours * wage_rate);

    cout << "\n_____________________________________________________\n";
    cout << "Pay to the order of " << get_name( ) << endl;
    cout << "The sum of " << get_net_pay( ) << " Dollars\n";
    cout << "_____________________________________________________\n";
    cout << "Check Stub: NOT NEGOTIABLE\n";
    cout << "Employee Number: " << get_ssn( ) << endl;
    cout << "Hourly Employee. \nHours worked: " << hours
         << " Rate: " << wage_rate << " Pay: " << get_net_pay( ) << endl;
    cout << "_____________________________________________________\n";
}


}//employeessavitch
```

list it in the class definition and give it a new definition, just as you would do with a member function that is added in the derived class. This is illustrated by the redefined function `print_check( )` of the class `HourlyEmployee` (Displays 16.3 and 16.5).

`SalariedEmployee` is another example of a derived class of the class `Employee`. The interface for the class `SalariedEmployee` is given in Display 16.4. An object declared to be of type `SalariedEmployee` has all the member functions and member variables of `Employee` and the new members given in the definition of the class `SalariedEmployee`. This is true even though the class `SalariedEmployee` lists none of the inherited variables and only lists one function from the class `Employee`, namely, the function `print_check`, which will have its definition changed in `SalariedEmployee`. The class `SalariedEmployee`, nonetheless, has the three member variables `name`, `ssn`, and `net_pay`, as well as the member variable `salary`. Notice that you do not have to declare the member variables and member functions of the class `Employee`, such as `name` and `set_name`, in order for a `SalariedEmployee` to have these members. The class `SalariedEmployee` gets these inherited members automatically without the programmer doing anything.

Note that the class `Employee` has all the code that is common to the two classes `HourlyEmployee` and `SalariedEmployee`. This saves you the trouble of writing identical code two times, once for the class `HourlyEmployee` and once for the class `SalariedEmployee`. Inheritance allows you to reuse the code in the class `Employee`.

An Object of a Derived Class Has More Than One Type

In everyday experience an hourly employee is an employee. In C++ the same sort of thing holds. Since `HourlyEmployee` is a derived class of the class `Employee`, every object of the class `HourlyEmployee` can be used anyplace an object of the class `Employee` can be used. In particular, you can use an argument of type `HourlyEmployee` when a function requires an argument of type `Employee`. You can assign an object of the class `HourlyEmployee` to a variable of type `Employee`. (But be warned: You cannot assign a plain old `Employee` object to a variable of type `HourlyEmployee`. After all, an `Employee` is not necessarily an `HourlyEmployee`.) Of course, the same remarks apply to any base class and its derived class. You can use an object of a derived class anyplace that an object of its base class is allowed.

More generally, an object of a class type can be used anyplace that an object of any of its ancestor classes can be used. If class `Child` is derived from class `Ancestor` and class `Grandchild` is derived from class `Child`, then an object of class `Grandchild` can be used anyplace an object of class `Child` can be used, and the object of class `Grandchild` can also be used anyplace that an object of class `Ancestor` can be used.

Constructors in Derived Classes

A constructor in a base class is not inherited in the derived class, but you can invoke a constructor of the base class within the definition of a derived class constructor, and that is all you need or normally want. A constructor for a derived class uses a constructor from the base class in a special way. A constructor for the base class initializes all the data inherited from the base class. Thus, a constructor for a derived class begins with an invocation of a constructor for the base class.

There is a special syntax for invoking the base class constructor that is illustrated by the constructor definitions for the class `HourlyEmployee` given in Display 16.5. In what follows we have reproduced (with minor changes in the line breaks to make it fit the text column) one of the constructor definitions for the class `HourlyEmployee` taken from that display:

```
HourlyEmployee::HourlyEmployee(string the_name,
     string the_number, double the_wage_rate,
     double the_hours)
     : Employee(the_name, the_number),
                 wage_rate(the_wage_rate), hours(the_hours)
{
    //deliberately empty
}
```

The portion after the colon is the initialization section of the constructor definition for the constructor `HourlyEmployee::HourlyEmployee`. The part `Employee(the_name, the_number)` is an invocation of the two-argument constructor for the base class `Employee`. Note that the syntax for invoking the base class constructor is analogous to the syntax used to set member variables: The entry `wage_rate(the_wage_rate)` sets the value of the member variable `wage_rate` to `the_wage_rate`; the entry `Employee(the_name, the_number)` invokes the base class constructor `Employee` with the arguments `the_name` and `the_number`. Since all the work is done in the initialization section, the body of the constructor definition is empty.

Below we reproduce the other constructor for the class `HourlyEmployee` from Display 16.5:

```
HourlyEmployee::HourlyEmployee( ) : Employee( ), wage_rate(0),
                                                   hours(0)
{
    //deliberately empty
}
```

In this constructor definition the default (zero-argument) version of the base class con-structor is called to initialize the inherited member variables. You should always

include an invocation of one of the base class constructors in the initialization section of a derived class constructor.

If a constructor definition for a derived class does not include an invocation of a constructor for the base class, then the default (zero-argument) version of the base class constructor will be invoked automatically. So, the following definition of the default constructor for the class `HourlyEmployee` (with `Employee( )` omitted) is equivalent to the version we just discussed:

```
HourlyEmployee::HourlyEmployee( ) : wage_rate(0), hours(0)
{
    //deliberately empty
}
```

However, we prefer to always explicitly include a call to a base class constructor, even if it would be invoked automatically.

A derived class object has all the member variables of the base class. When a derived class constructor is called, these member variables need to be allocated memory and should be initialized. This allocation of memory for the inherited member variables must be done by a constructor for the base class, and the base class constructor is the most convenient place to initialize these inherited member variables. That is why you should always include a call to one of the base class constructors when you define a constructor for a derived class. If you do not include a call to a base class constructor (in the initialization section of the definition of a derived class constructor), then the default (zero-argument) constructor of the base class is called automatically. (If there is no default constructor for the base class, that is an error condition.)

order of constructor calls

The call to the base class constructor is the first action taken by a derived class constructor. Thus, if class B is derived from class A and class C is derived from class B, then when an object of the class C is created, first a constructor for the class A is called, then a constructor for B is called, and finally the remaining actions of the C constructor are taken.

Constructors in Derived Classes

A derived class does not inherit the constructors of its base class. However, when defining a constructor for the derived class, you can and should include a call to a constructor of the base class (within the initialization section of the constructor definition).

If you do not include a call to a constructor of the base class, then the default (zero-argument) constructor of the base class will automatically be called when the derived class constructor is called.

PITFALL Use of Private Member Variables from the Base Class

An object of the class `HourlyEmployee` (Displays 16.3 and 16.5) inherits a member variable called `name` from the class `Employee` (Displays 16.1 and 16.2). For example, the following code would set the value of the member variable `name` of the object `joe` to `"Josephine"`. (This code also sets the member variable `ssn` to `"123-45-6789"` and both the `wage_rate` and `hours` to 0.)

```
HourlyEmployee joe("Josephine", "123-45-6789", 0, 0);
```

If you want to change `joe.name` to `"Mighty-Joe"` you can do so as follows:

```
joe.set_name("Mighty-Joe");
```

But, you must be a bit careful about how you manipulate inherited member variables such as `name`. The member variable `name` of the class `HourlyEmployee` was inherited from the class `Employee`, but the member variable `name` is a private member variable in the definition of the class `Employee`. That means that `name` can only be directly accessed within the definition of a member function in the class `Employee`. A member variable (or member function) that is private in a base class is not accessible *by name* in the definition of a member function for *any other class, not even in a member function definition of a derived class.* Thus, although the class `HourlyEmployee` does have a member variable named `name` (inherited from the base class `Employee`), it is illegal to directly access the member variable `name` in the definition of any member function in the class definition of `HourlyEmployee`.

For example, the following are the first few lines from the body of the member function `HourlyEmployee::print_check` (taken from Display 16.5):

```
void HourlyEmployee::print_check( )
{
    set_net_pay(hours * wage_rate);

    cout << "\n_____\n";
    cout << "Pay to the order of " << get_name( ) << endl;
    cout << "The sum of " << get_net_pay( ) << " Dollars\n";
```

You might have wondered why we needed to use the member function `set_net_pay` to set the value of the `net_pay` member variable. You might be tempted to rewrite the start of the member function definition as follows:

```
void HourlyEmployee::print_check( )
{
   net_pay = hours * wage_rate;          Illegal use of net_pay
```

As the comment indicates, this will not work. The member variable `net_pay` is a private member variable in the class `Employee`, and although a derived class like `HourlyEmployee` inherits the variable `net_pay`, it cannot access it directly. It must use some public member function to access the member variable `net_pay`. The correct way to accomplish the definition of `print_check` in the class `HourlyEmployee` is the way we did it in Display 16.5 (and part of which was displayed earlier).

The fact that `name` and `net_pay` are inherited variables that are private in the base class also explains why we needed to use the accessor functions `get_name` and `get_net_pay` in the definition of `HourlyEmployee::print_check` instead of simply using the variable names `name` and `net_pay`. You cannot mention a private inherited member variable by name. You must instead use public accessor and mutator member functions (such as `get_name` and `set_name`) that were defined in the base class. (Recall that an *accessor function* is a function that allows you to access member variables of a class, and a *mutator function* is one that allows you to change member variables of a class. Accessor and mutator functions were covered in Chapter 6.)

The fact that a private member variable of a base class cannot be accessed in the definition of a member function of a derived class often seems wrong to people. After all, if you are an hourly employee and you want to change your name, nobody says, "Sorry `name` is a private member variable of the class `Employee`." After all, if you are an hourly employee, you are also an employee. In Java, this is also true; an object of the class `HourlyEmployee` is also an object of the class `Employee`. However, the laws on the use of private member variables and member functions must be as we described, or else their privacy would be compromised. If private member variables of a class were accessible in member function definitions of a derived class, then anytime you wanted to access a private member variable, you could simply create a derived class and access it in a member function of that class, which would mean that all private member variables would be accessible to anybody who wanted to put in a little extra effort. This adversarial scenario illustrates the problem, but the big problem is unintentional errors, not intentional subversion. If private member variables of a class were accessible in member function definitions of a derived class, then the member variables might be changed by mistake or in inappropriate ways. (Remember, accessor and mutator functions can guard against inappropriate changes to member variables.)

We will discuss one possible way to get around this restriction on private member variables of the base class in the subsection entitled "The *protected* Qualifier" a bit later in this chapter.

PITFALL Private Member Functions Are Effectively Not Inherited

As we noted in the previous Pitfall section, a member variable (or member function) that is private in a base class is not directly accessible outside of the interface and implementation of the base class, *not even in a member function definition for a derived class*. Note that private member functions are just like private variables in terms of not being directly available. But in the case of member functions, the restriction is more dramatic. A private variable can be accessed indirectly via an accessor or mutator member function. A private member function is simply not available. It is just as if the private member function were not inherited.

This should not be a problem. Private member functions should just be used as helping functions, and so their use should be limited to the class in which they are defined. If you want a member function to be used as a helping member function in a number of inherited classes, then it is not *just* a helping function, and you should make the member function public.

The *protected* Qualifier

As you have seen, you cannot access a private member variable or private member function in the definition or implementation of a derived class. There is a classification of member variables and functions that allows them to be accessed by name in a derived class but not anyplace else, such as in some class that is not a derived class. If you use the qualifier *protected*, rather than *private* or *public*, before a member variable or member function of a class, then for any class or function other than a derived class, the effect is the same as if the member variable were labeled *private*; however, in a derived class the variable can be accessed by name.

protected

For example, consider the class `HourlyEmployee` that was derived from the base class `Employee`. We were required to use accessor and mutator member functions to manipulate the inherited member variables in the definition of `HourlyEmployee::print_check`. If all the private member variables in the class `Employee` were labeled with the keyword *protected* instead of *private*, the definition of `HourlyEmployee::print_check` in the derived class `Employee` could be simplified to the following:

```
void HourlyEmployee::print_check( )
//Only works if the member variables of Employee are marked
//protected instead of private.
{
    net_pay = hours * wage_rate;

    cout << "\n_____________________________________________\n";
    cout << "Pay to the order of " << name << endl;
    cout << "The sum of " << net_pay << " Dollars\n";
    cout << "_____________________________________________\n";
    cout << "Check Stub: NOT NEGOTIABLE\n";
    cout << "Employee Number: " << ssn << endl;
    cout << "Hourly Employee. \nHours worked: " << hours
         << " Rate: " << wage_rate << " Pay: " << net_pay
         << endl;
    cout << "_____________________________________________\n";
}
```

In the derived class `HourlyEmployee`, the inherited member variables `name`, `net_pay`, and `ssn` can be accessed by name provided they are marked as *`protected`* (as opposed to *`private`*) in the base class `Employee`. However, in any class that is not derived from the class `Employee`, these member variables are treated as if they were marked *`private`*.

Member variables that are protected in the base class act as though they were also marked *`protected`* in any derived class. For example, suppose you define a derived class `PartTimeHourlyEmployee` of the class `HourlyEmployee`. The class `PartTimeHourlyEmployee` inherits all the member variables of the class `HourlyEmployee`, including the member variables that `HourlyEmployee` inherits from the class `Employee`. So, the class `PartTimeHourlyEmployee` will have the member variables `net_pay`, `name`, and `ssn`. If these member variables were marked *`protected`* in the class `Employee`, then they can be used by name in the definitions of functions of the class `PartTimeHourlyEmployee`.

Except for derived classes (and derived classes of derived classes, etc.) a member variable that is marked *`protected`* is treated the same as if it were marked *`private`*.

We include a discussion of *`protected`* member variables primarily because you will see them used and should be familiar with them. Many, but not all, programming authorities say it is bad style to use *`protected`* member variables. They say it compromises the principle of hiding the class implementation. They say that all member variables should be marked *`private`*. If all member variables are marked *`private`*, the inherited member variables cannot be accessed by name in derived class function definitions. However, this is not as bad as its sounds. The inherited

private member variables can be accessed indirectly by invoking inherited functions that either read or change the *private* inherited variables. Since authorities differ on whether or not you should use protected members, you will have to make your own decision on whether or not to use them.

Protected Members

If you use the qualifier *protected*, rather than *private* or *public*, before a member variable of a class, then for any class or function other than a derived class (or a derived class of a derived class, etc.) the situation is the same as if the member variable were labeled *private*. However, in the definition of a member function of a derived class, the variable can be accessed by name. Similarly, if you use the qualifier *protected* before a member function of a class, then for any class or function other than a derived class (or a derived class of a derived class, etc.), that is the same as if the member function were labeled *private*. However, in the definition of a member function of a derived class the protected function can be used.

Inherited protected members are inherited in the derived class as if they were marked *protected* in the derived class. In other words, if a member is marked as *protected* in a base class, then it can be accessed by name in the definitions of all descendant classes, not just in those classes directly derived from the base class.

SELF-TEST EXERCISES

1 Is the following program legal (assuming appropriate `#include` and *using* directives are added)?

```
void show_employee_data(const Employee object);

int main( )
{
    HourlyEmployee joe("Mighty Joe",
                              "123-45-6789", 20.50, 40);
    SalariedEmployee boss("Mr. Big Shot",
                              "987-65-4321", 10500.50);
    show_employee_data(joe);
    show_employee_data(boss);

    return 0;
}
```

```
void show_employee_data(const Employee object)
{
    cout << "Name: " << object.get_name( ) << endl;
    cout << "Social Security Number: "
         << object.get_ssn( ) << endl;
}
```

2 Give a definition for a class `SmartBut` that is a derived class of the base class `Smart`, which we reproduce for you here. Do not bother with `#include` directives or namespace details.

```
class Smart
{
public:
    Smart( );
    void print_answer( ) const;
protected:
    int a;
    int b;
};
```

This class should have an additional data field, `crazy`, that is of type *bool*, one additional member function that takes no arguments and returns a value of type *bool*, and suitable constructors. The new function is named `is_crazy`. You do not need to give any implementations, just the class definition.

3 Is the following a legal definition of the member function `is_crazy` in the derived class `SmartBut` discussed in Self-Test Exercise 2? Explain your answer. (Remember, the question asks if it is legal, not if it is a sensible definition.)

```
bool SmartBut::is_crazy( ) const
{
   if (a > b)
       return crazy;
   else
       return true;
}
```

Redefinition of Member Functions

In the definition of the derived class `HourlyEmployee` (Display 16.3), we gave the declarations for the new member functions `set_rate`, `get_rate`, `set_hours`, and `get_hours`. We also gave the function declaration for only one of the member functions inherited from the class `Employee`. The inherited member functions whose

function declarations were not given (such as `set_name` and `set_ssn`) are inherited unchanged. They have the same definition in the class `HourlyEmployee` as they do in the base class `Employee`. When you define a derived class like `HourlyEmployee`, you only list the function declarations for the inherited member functions whose definitions you want to change to have a different definition in the derived class. If you look at the implementation of the class `HourlyEmployee`, given in Display 16.5, you will see that we have redefined the inherited member function `print_check`. The class `SalariedEmployee` also gives a new definition to the member function `print_check`, as shown in Display 16.6. Moreover, the two classes give different definitions from each other. The function `print_check` is **redefined** in the derived classes.

redefined function

Display 16.7 gives a demonstration program that illustrates the use of the derived classes `HourlyEmployee` and `SalariedEmployee`.

Redefining an Inherited Function

A derived class inherits all the member functions (and member variables as well) that belong to the base class. However, if a derived class requires a different implementation for an inherited member function, the function may be redefined in the derived class. When a member function is redefined, you must list its declaration in the definition of the derived class even though the declaration is the same as in the base class. If you do not wish to redefine a member function that is inherited from the base class, then it is not listed in the definition of the derived class.

Redefining versus Overloading

Do not confuse *redefining* a function definition in a derived class with *overloading* a function name. When you redefine a function definition, the new function definition given in the derived class has the same number and types of parameters. On the other hand, if the function in the derived class were to have a different number of parameters or a parameter of a different type from the function in the base class, then the derived class would have both functions. That would be overloading. For example, suppose we added a function with the following function declaration to the definition of the class `HourlyEmployee`:

```
void set_name(string first_name, string last_name);
```

The class `HourlyEmployee` would have this two-argument function `set_name`, and it would also inherit the following one-argument function `set_name`:

```
void set_name(string new_name);
```

Display 16.6 Implementation for the Derived Class `SalariedEmployee` (*part 1 of 2*)

```
//This is the file salariedemployee.cpp.
//This is the implementation for the class SalariedEmployee.
//The interface for the class SalariedEmployee is in
//the header file salariedemployee.h.
#include <iostream>
#include <string>
#include "salariedemployee.h"
using namespace std;

namespace employeessavitch
{
    SalariedEmployee::SalariedEmployee( ) : Employee( ), salary(0)
    {
        //deliberately empty
    }

    SalariedEmployee::SalariedEmployee(string the_name, string the_number,
                                       double the_weekly_salary)
                    : Employee(the_name, the_number), salary(the_weekly_salary)
    {
        //deliberately empty
    }

    double SalariedEmployee::get_salary( ) const
    {
        return salary;
    }

    void SalariedEmployee::set_salary(double new_salary)
    {
        salary = new_salary;
    }
```

Display 16.6 Implementation for the Derived Class `SalariedEmployee` (*part 2 of 2*)

```
    void SalariedEmployee::print_check( )
    {
        set_net_pay(salary);
        cout << "\n_____________________________________________________\n";
        cout << "Pay to the order of " << get_name( ) << endl;
        cout << "The sum of " << get_net_pay( ) << " Dollars\n";
        cout << "_____________________________________________________\n";
        cout << "Check Stub NOT NEGOTIABLE \n";
        cout << "Employee Number: " << get_ssn( ) << endl;
        cout << "Salaried Employee. Regular Pay: "
             << salary << endl;
        cout << "_____________________________________________________\n";
    }
}//employeessavitch
```

The class `HourlyEmployee` would have two functions named `set_name`. This would be *overloading* the function name `set_name`.

On the other hand, both the class `Employee` and the class `HourlyEmployee` define a function with the following function declaration:

```
void print_check( );
```

In this case, the class `HourlyEmployee` has only one function named `print_check`, but the definition of the function `print_check` for the class `HourlyEmployee` is different from its definition for the class `Employee`. In this case, the function `print_check` has been *redefined*.

If you get redefining and overloading confused, you do have one consolation. They are both legal. So, it is more important to learn how to use them than it is to learn to distinguish between them. Nonetheless, you should learn the difference between them.

Access to a Redefined Base Function

Suppose you redefine a function so that it has a different definition in the derived class from what it had in the base class. The definition that was given in the base class is not completely lost to the derived class objects. However, if you want to invoke the version of the function given in the base class with an object in the derived class, you need some way to say "use the definition of this function as given

Display 16.7 Using Derived Classes (*part 1 of 2*)


```
#include <iostream>
#include "hourlyemployee.h"
#include "salariedemployee.h"
using std::cout;
using std::endl;
using namespace employeessavitch;

int main( )
{
    HourlyEmployee joe;
    joe.set_name("Mighty Joe");
    joe.set_ssn("123-45-6789");
    joe.set_rate(20.50);
    joe.set_hours(40);
    cout << "Check for " << joe.get_name( )
         << " for " << joe.get_hours( ) << " hours.\n";
    joe.print_check( );
    cout << endl;

    SalariedEmployee boss("Mr. Big Shot", "987-65-4321", 10500.50);
    cout << "Check for " << boss.get_name( ) << endl;
    boss.print_check( );

    return 0;
}
```

The functions `set_name`, `set_ssn`, `set_rate`, `set_hours`, *and* `get_name` *are inherited unchanged from the class* `Employee`.
The function `print_check` *is redefined.*
The function `get_hours` *was added to the derived class* `HourlyEmployee`.

Display 16.7 Using Derived Classes (*part 2 of 2*)

Sample Dialogue

```
Check for Mighty Joe for 40 hours.


_________________________________________________
Pay to the order of Mighty Joe
The sum of 820 Dollars
_________________________________________________
Check Stub: NOT NEGOTIABLE
Employee Number: 123-45-6789
Hourly Employee.
Hours worked: 40 Rate: 20.5 Pay: 820
_________________________________________________

Check for Mr. Big Shot


_________________________________________________
Pay to the order of Mr. Big Shot
The sum of 10500.5 Dollars
_________________________________________________
Check Stub NOT NEGOTIABLE
Employee Number: 987-65-4321
Salaried Employee. Regular Pay: 10500.5
_________________________________________________
```

in the base class (even though I am an object of the derived class)." The way you say this is to use the scope resolution operator with the name of the base class. An example should clarify the details.

Consider the base class `Employee` (Display 16.1) and the derived class `HourlyEmployee` (Display 16.3). The function `print_check( )` is defined in both classes. Now suppose you have an object of each class, as in

```
Employee jane_e;
HourlyEmployee sally_h;
```

Signature

A function's **signature** is the function's name with the sequence of types in the parameter list, not including the *const* keyword and not including the ampersand (&). When you overload a function name, the two definitions of the function name must have different signatures using this definition of signature.[2]

If a function has the same name in a derived class as in the base class but has a different signature, that is overloading, not redefinition.

Then

```
jane_e.print_check( );
```

uses the definition of `print_check` given in the class `Employee`, and

```
sally_h.print_check( );
```

uses the definition of `print_check` given in the class `HourlyEmployee`.

But, suppose you want to invoke the version of `print_check` given in the definition of the base class `Employee` with the derived class object `sally_h` as the calling object for `print_check`. You do that as follows:

```
sally_h.Employee::print_check( );
```

Of course, you are unlikely to want to use the version of `print_check` given in the particular class `Employee`, but with other classes and other functions, you may occasionally want to use a function definition from a base class with a derived class object. An example is given in Self-Test Exercise 6.

SELF-TEST EXERCISES

4 The class `SalariedEmployee` inherits both of the functions `get_name` and `print_check` (among other things) from the base class `Employee`, yet only the function declaration for the function `print_check` is given in the definition of the class `SalariedEmployee`. Why isn't the function declaration for the function `get_name` given in the definition of `SalariedEmployee`?

[2]Some compilers may allow overloading on the basis of *const* versus no *const*, but you cannot count on this and so should not do it. For this reason some definitions of *signature* include the *const* modifier, but this is a cloudy issue that is best avoided until you become an expert.

5 Give a definition for a class `TitledEmployee` that is a derived class of the base class `SalariedEmployee` given in Display 16.4. The class `TitledEmployee` has one additional member variable of type `string` called `title`. It also has two additional member functions: `get_title`, which takes no arguments and returns a `string`; and `set_title`, which is a *`void`* function that takes one argument of type `string`. It also redefines the member function `set_name`. You do not need to give any implementations, just the class definition. However, do give all needed `#include` directives and all *`using`* `namespace` directives. Place the class `TitledEmployee` in the namespace `employeessavitch`.

6 Give the definitions of the constructors for the class `TitledEmployee` that you gave as the answer to Self-Test Exercise 5. Also, give the redefinition of the member function `set_name`. The function `set_name` should insert the title into the name. Do not bother with `#include` directives or namespace details.

16.2 Inheritance Details

The devil is in the details.

COMMON SAYING

This section presents some of the more subtle details about inheritance. Most of the topics are only relevant to classes that use dynamic arrays or pointers and other dynamic data.

Functions That Are Not Inherited

As a general rule if `Derived` is a derived class with base class `Base`, then all "normal" functions in the class `Base` are inherited members of the class `Derived`. However, there are some special functions that are, for all practical purposes, not inherited. We have already seen that, as a practical matter, constructors are not inherited and that private member functions are not inherited. Destructors are also effectively not inherited.

In the case of the copy constructor, it is not inherited, but if you do not define a copy constructor in a derived class (or any class for that matter), C++ will automatically generate a copy constructor for you. However, this default copy constructor simply copies the contents of member variables and does not work correctly for classes with pointers or dynamic data in their member variables. Thus, if your class member variables involve pointers, dynamic arrays, or other dynamic data, then you should define a copy constructor for the class. This applies whether or not the class is a derived class.

The assignment operator = is also not inherited. If the base class `Base` defines the assignment operator, but the derived class `Derived` does not define the assignment operator, then the class `Derived` will have an assignment operator, but it will be the default assignment operator that C++ creates (when you do not define =); it will not have anything to do with the base class assignment operator defined in `Base`.

It is natural that constructors, destructors, and the assignment operator are not inherited. To correctly perform their tasks they need information that the base class does not possess. To correctly perform their functions, they need to know about the new member variables introduced in the derived class.

Assignment Operators and Copy Constructors in Derived Classes

Overloaded assignment operators and constructors are not inherited. However, they can be, and in almost all cases must be, used in the definitions of overloaded assignment operators and copy constructors in derived classes.

When overloading the assignment operator in a derived class, you normally use the overloaded assignment operator from the base class. We will present an outline of how the code for doing this is written. To help understand the code outline, remember that an overloaded assignment operator must be defined as a member function of the class.

If `Derived` is a class derived from `Base`, then the definition of the overloaded assignment operator for the class `Derived` would typically begin with something like the following:

```
Derived& Derived::operator =(const Derived& right_side)
{
    Base::operator =(right_side);
```

The first line of code in the body of the definition is a call to the overloaded assignment operator of the `Base` class. This takes care of the inherited member variables and their data. The definition of the overloaded assignment operator would then go on to set the new member variables that were introduced in the definition of the class `Derived`.

A similar situation holds for defining the copy constructor in a derived class. If `Derived` is a class derived from `Base`, then the definition of the copy constructor for the class `Derived` would typically use the copy constructor for the class `Base` to set up the inherited member variables and their data. The code would typically begin with something like the following:

```
Derived::Derived(const Derived& object)
                    : Base(object), <probably more initializations>
{
```

The invocation of the base class copy constructor `Base(object)` sets up the inherited member variables of the `Derived` class object being created. Note that since `object` is of type `Derived`, it is also of type `Base`; therefore, `object` is a legal argument to the copy constructor for the class `Base`.

Of course, these techniques do not work unless you have a correctly functioning assignment operator and a correctly functioning copy constructor for the base class. This means that the base class definition must include a copy constructor and that either the default automatically created assignment operator must work correctly for the base class or the base class must have a suitable overloaded definition of the assignment operator.

Destructors in Derived Classes

If a base class has a correctly functioning destructor, then it is relatively easy to define a correctly functioning destructor in a class derived from the base class. When the destructor for the derived class is invoked, it automatically invokes the destructor of the base class, so there is no need for the explicit writing of a call to the base class destructor; it always happens automatically. The derived class destructor therefore need only worry about using *`delete`* on the member variables (and any data they point to) that are added in the derived class. It is the job of the base class destructor to invoke *`delete`* on the inherited member variables.

If class B is derived from class A and class C is derived from class B, then when an object of the class C goes out of scope, first the destructor for the class C is called, then the destructor for class B is called, and finally the destructor for class A is called. Note that the order in which destructors are called is the reverse of the order in which constructors are called.

SELF-TEST EXERCISES

7 You know that an overloaded assignment operator and a copy constructor are not inherited. Does this mean that if you do not define an overloaded assignment operator or a copy constructor for a derived class, then that derived class will have no assignment operator and no copy constructor?

8 Suppose `Child` is a class derived from the class `Parent`, and the class `Grandchild` is a class derived from the class `Child`. This question is concerned with the constructors and destructors for the three classes `Parent`, `Child`, and `Grandchild`. When a constructor for the class `Grandchild` is invoked, what constructors are invoked and in what order? When the destructor for the class `Grandchild` is invoked, what destructors are invoked and in what order?

9 Give the definitions for the member function add_value, the copy constructor, the overloaded assignment operator, and the destructor for the following class. This class is intended to be a class for a partially filled array. The member variable number_used contains the number of array positions currently filled. The other constructor definition is given to help you get started.

```
#include <iostream>
#include <cstdlib>
using namespace std;

class PartFilledArray
{
public:
    PartFilledArray(int array_size);
    PartFilledArray(const PartFilledArray& object);
    ~PartFilledArray();
    void operator = (const PartFilledArray& right_side);
    void add_value(double new_entry);
    //There would probably be more member functions
    //but they are irrelevant to this exercise.
protected:
    double *a;
    int max_number;
    int number_used;
};

PartFilledArray::PartFilledArray(int array_size)
              : max_number(array_size), number_used(0)
{
    a = new double[max_number];
}
```

(Many authorities would say that the member variables should be private rather than protected. We tend to agree. However, using *protected* makes for a better practice assignment, and you should have some experience with protected variables because some programmers do use them.)

10 Define a class called PartFilledArrayWMax that is a derived class of the class PartFilledArray. The class PartFilledArrayWMax has one additional member variable named max_value that holds the maximum value stored in the array. Define a member accessor function named get_max that returns the maximum value stored in the array. Redefine the member function add_value

and define two constructors, one of which has an `int` argument for the maximum number of entries in the array. Also define a copy constructor, an overloaded assignment operator, and a destructor. (A real class would have more member functions, but these will do for an exercise.)

16.3 Polymorphism

I did it my way.

FRANK SINATRA

polymorphism
function

Polymorphism refers to the ability to associate multiple meanings to one function name. As it has come to be used today *polymorphism* refers to a very particular way of associating multiple meanings to a single function name. That is, **polymorphism** refers to the ability to associate multiple meanings to one function name by means of a special mechanism known as *late binding*. Polymorphism is one of the key components of a programming philosophy known as *object-oriented programming*. Late binding, and therefore polymorphism, is the topic of this section.

Late Binding

A *virtual function* is one that, in some sense, may be used before it is defined. For example, a graphics program may have several kinds of figures, such as rectangles, circles, ovals, and so forth. Each figure might be an object of a different class. For example, the `Rectangle` class might have member variables for a height, width, and center point, while the `Circle` class might have member variables for a center point and a radius. In a well-designed programming project, all of them would probably be descendants of a single parent class called, for example, `Figure`. Now, suppose you want a function to draw a figure on the screen. To draw a circle, you need different instructions from those you need to draw a rectangle. So, each class needs to have a different function to draw its kind of figure. However, because the functions belong to the classes, they can all be called `draw`. If `r` is a `Rectangle` object and `c` is a `Circle` object, then `r.draw()` and `c.draw()` can be functions implemented with different code. All this is not news, but now we move on to something new: *virtual functions* defined in the parent class `Figure`.

Now, the parent class `Figure` may have functions that apply to all figures. For example, it might have a function called `center` that moves a figure to the center of the screen by erasing it and then redrawing it in the center of the screen. `Figure::center` might use the function `draw` to redraw the figure in the center of the screen. When you think of using the inherited function `center` with figures of the classes `Rectangle` and `Circle`, you begin to see that there are complications here.

To make the point clear and more dramatic, let's suppose the class `Figure` is already written and in use and at some later time we add a class for a brand-new kind of figure, say, the class `Triangle`. Now, `Triangle` can be a derived class of the class `Figure`, and so the function `center` will be inherited from the class `Figure`; thus, the function `center` should apply to (and perform correctly for!) all `Triangle`s. But there is a complication. The function `center` uses `draw`, and the function `draw` is different for each type of figure. The inherited function `center` (if nothing special is done) will use the definition of the function `draw` given in the class `Figure`, and that function `draw` does not work correctly for `Triangle`s. We want the inherited function `center` to use the function `Triangle::draw` rather than the function `Figure::draw`. But the class `Triangle`, and therefore the function `Triangle::draw`, was not even written when the function `center` (defined in the class `Figure`) was written and compiled! How can the function `center` possibly work correctly for `Triangle`s? The compiler did not know anything about `Triangle::draw` at the time that `center` was compiled. The answer is that it can apply provided `draw` is a *virtual function*.

virtual function

When you make a function **virtual,** you are telling the compiler "I do not know how this function is implemented. Wait until it is used in a program, and then get the implementation from the object instance." The technique of waiting until runtime to determine the implementation of a procedure is called **late binding** or **dynamic binding.** Virtual functions are the way C++ provides late binding. But enough introduction. We need an example to make this come alive (and to teach you how to use virtual functions in your programs). In order to explain the details of virtual functions in C++, we will use a simplified example from an application area other than drawing figures.

late binding
dynamic binding

Virtual Functions in C++

Suppose you are designing a record-keeping program for an automobile parts store. You want to make the program versatile, but you are not sure you can account for all possible situations. For example, you want to keep track of sales, but you cannot anticipate all types of sales. At first, there will only be regular sales to retail customers who go to the store to buy one particular part. However, later you may want to add sales with discounts, or mail-order sales with a shipping charge. All these sales will be for an item with a basic price and ultimately will produce some bill. For a simple sale, the bill is just the basic price, but if you later add discounts, then some kinds of bills will also depend on the size of the discount. Your program will need to compute daily gross sales, which intuitively should just be the sum of all the individual sales bills. You may also want to calculate the largest and smallest sales of the day or the average sale for the day. All these can be calculated from the individual bills, but the functions for computing the bills will not be added until later, when you

decide what types of sales you will be dealing with. To accommodate this, we make the function for computing the bill a virtual function. (For simplicity in this first example, we assume that each sale is for just one item, although with derived classes and virtual functions we could, but will not here, account for sales of multiple items.)

Displays 16.8 and 16.9 contain the interface and implementation for the class `Sale`. All types of sales will be derived classes of the class `Sale`. The class `Sale` corresponds to simple sales of a single item with no added discounts or charges. Notice the reserved word *`virtual`* in the function declaration for the function `bill` (Display 16.8). Notice (Display 16.9) that the member function `savings` and the overloaded operator < both use the function `bill`. Since `bill` is declared to be a virtual function, we can later define derived classes of the class `Sale` and define their versions of the function `bill`, and the definitions of the member function `savings` and the overloaded operator <, which we gave with the class `Sale`, will use the version of the function `bill` that corresponds to the object of the derived class.

For example, Display 16.10 shows the derived class `DiscountSale`. Notice that the class `DiscountSale` requires a different definition for its version of the function `bill`. Nonetheless, when the member function `savings` and the overloaded operator < are used with an object of the class `DiscountSale`, they will use the version of the function definition for `bill` that was given with the class `DiscountSale`. This is indeed a pretty fancy trick for C++ to pull off. Consider the function call `d1.savings(d2)` for objects `d1` and `d2` of the class `DiscountSale`. The definition of the function `savings` (even for an object of the class `DiscountSale`) is given in the implementation file for the base class `Sale`, which was compiled before we ever even thought of the class `DiscountSale`. Yet, in the function call `d1.savings(d2)`, the line that calls the function `bill` knows enough to use the definition of the function `bill` given for the class `DiscountSale`.

How does this work? In order to write C++ programs you can just assume it happens by magic, but the real explanation was given in the introduction to this section. When you label a function *`virtual`*, you are telling the C++ environment "Wait until this function is used in a program, and then get the implementation corresponding to the calling object."

Display 16.11 gives a sample program that illustrates how the virtual function `bill` and the functions that use `bill` work in a complete program.

There are a number of technical details you need to know in order to use virtual functions in C++. We list them in what follows:

- If a function will have a different definition in a derived class than in the base class and you want it to be a virtual function, you add the keyword *`virtual`* to the function declaration in the base class. You do not need to add the reserved word *`virtual`* to the function declaration in the derived class. If a function is virtual in the base class, then it is automatically virtual in the derived class. (However, it is a good idea to label the function declaration in the derived class *`virtual`*, even though it is not required.)

Display 16.8 Interface for the Base Class `Sale`

```
//This is the header file sale.h.
//This is the interface for the class Sale.
//Sale is a class for simple sales.
#ifndef SALE_H
#define SALE_H

#include <iostream>
using namespace std;

namespace salesavitch
{

    class Sale
    {
    public:
        Sale();
        Sale(double the_price);
        virtual double bill() const;
        double savings(const Sale& other) const;
        //Returns the savings if you buy other instead of the calling object.
    protected:
        double price;
    };

    bool operator < (const Sale& first, const Sale& second);
    //Compares two sales to see which is larger.

}//salesavitch

#endif // SALE_H
```

Display 16.9 Implementation of the Base Class `Sale`

```
//This is the implementation file: sale.cpp
//This is the implementation for the class Sale.
//The interface for the class Sale is in
//the header file sale.h.
#include "sale.h"

namespace salesavitch
{

    Sale::Sale() : price(0)
    {}

    Sale::Sale(double the_price) : price(the_price)
    {}

    double Sale::bill() const
    {
        return price;
    }

    double Sale::savings(const Sale& other) const
    {
        return ( bill() - other.bill() );
    }

    bool operator < (const Sale& first, const Sale& second)
    {
        return (first.bill() < second.bill());
    }

}//salesavitch
```

Display 16.10 The Derived Class `DiscountSale`

This is the file `discountsale.h`.

```
//This is the interface for the class DiscountSale.
#ifndef DISCOUNTSALE_H
#define DISCOUNTSALE_H
#include "sale.h"

namespace salesavitch
{
    class DiscountSale : public Sale
    {
    public:
        DiscountSale();
        DiscountSale(double the_price, double the_discount);
        //Discount is expressed as a percent of the price.
        virtual double bill() const;
    protected:
        double discount;
    };
}//salesavitch
#endif //DISCOUNTSALE_H
```

The keyword `virtual` *is not required here, but it is good style to include it.*

This is the file `discountsale.cpp`.

```
//This is the implementation for the class DiscountSale.
#include "discountsale.h"

namespace salesavitch
{
    DiscountSale::DiscountSale() : Sale(), discount(0)
    {}

    DiscountSale::DiscountSale(double the_price, double the_discount)
              : Sale(the_price), discount(the_discount)
    {}

    double DiscountSale::bill() const
    {
        double fraction = discount/100;
        return (1 - fraction)*price;
    }
}//salesavitch
```

Display 16.11 Use of a Virtual Function


```
//Demonstrates the performance of the virtual function bill.
#include <iostream>
#include "sale.h" //Not really needed, but safe due to ifndef.
#include "discountsale.h"
using namespace std;
using namespace salesavitch;

int main()
{
    Sale simple(10.00);//One item at $10.00.
    DiscountSale discount(11.00, 10);//One item at $11.00 with a 10% discount.

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);

    if (discount < simple)
    {
        cout << "Discounted item is cheaper.\n";
        cout << "Savings is $" << simple.savings(discount) << endl;
    }
    else
        cout << "Discounted item is not cheaper.\n";

    return 0;
}
```

Sample Dialogue

```
Discounted item is cheaper.
Savings is $0.10
```

- The reserved word `virtual` is added to the function declaration and not to the function definition.
- You do not get a virtual function and the benefits of virtual functions unless you use the keyword `virtual`.

Since virtual functions are so great, why not make all member functions virtual? Almost the only reason for not always using virtual functions is efficiency. The compiler and the run-time environment need to do much more work for virtual functions, and so if you label more member function `virtual` than you need to, your programs will be less efficient.

Overriding

When a virtual function definition is changed in a derived class, programmers often say the function definition is **overridden.** In the C++ literature, a distinction is sometimes made between the terms *redefined* and *overridden*. Both terms refer to changing the definition of the function in a derived class. If the function is a virtual function it's called *overriding*. If the function is not a virtual function, it's called *redefining*. This may seem like a silly distinction to you the programmer, since you do the same thing in both cases, but the two cases are treated differently by the compiler.

Polymorphism

The term **polymorphism** refers to the ability to associate multiple meanings to one function name by means of late binding. Thus, polymorphism, late binding, and virtual functions are really all the same topic.

SELF-TEST EXERCISE

11 Suppose you modify the definitions of the class `Sale` (Display 16.8) by deleting the reserved word `virtual`. How would that change the output of the program in Display 16.11?

Virtual Functions and Extended Type Compatibility

We will discuss some of the further consequences of declaring a class member function to be `virtual` and do one example that uses some of these features.

C++ is a fairly strongly typed language. This means that the types of items are always checked and an error message is issued if there is a type mismatch, such as a type mismatch between an argument and a formal parameter when there is no conversion that can be automatically invoked. This also means that normally the value assigned to a variable must match the type of the variable, although in a few well-defined cases C++ will perform an automatic type cast (called a coercion) so that it appears that you can assign a value of one type to a variable of another type.

For example, C++ allows you to assign a value of type *char* or *int* to a variable of type *double*. However, C++ does not allow you to assign a value of type *double* or *float* to a variable of any integer type (*char*, *short*, *int*, *long*).

However, as important as strong typing is, this strong type checking interferes with the very idea of inheritance in object-oriented programming. Suppose you have defined class A and class B and have defined objects of type class A and class B. You cannot always assign between objects of these types. For example, suppose a program or unit contains the following type declarations:

```
class Pet
{
public:
    virtual void print();
    string name;
};

class Dog : public Pet
{
public:
    virtual void print(); //keyword virtual not needed, but is
            //put here for clarity. (It is also good style!)

    string breed;
};

Dog vdog;
Pet vpet;
```

Now concentrate on the data members, name and breed. (To keep this example simple, we have made the member variables *public*. In a real application, they should be *private* and have functions to manipulate them.)

Anything that is a Dog is also a Pet. It would seem to make sense to allow programs to consider values of type Dog to also be values of type Pet, and hence the following should be allowed:

```
vdog.name = "Tiny";
vdog.breed = "Great Dane";
vpet = vdog;
```

C++ does allow this sort of assignment. You may assign a value, such as the value of `vdog`, to a variable of a parent type, such as `vpet`, but you are not allowed to perform the reverse assignment. Although the above assignment is allowed, the value that is assigned to the variable `vpet` loses its `breed` field. This is called the slicing problem. The following attempted access will produce an error message:

slicing problem

```
cout << vpet.breed;
            // Illegal: class Pet has no member named breed
```

You can argue that this makes sense, since once a `Dog` is moved to a variable of type `Pet` it should be treated like any other `Pet` and not have properties peculiar to `Dog`s. This makes for a lively philosophical debate, but it usually just makes for a nuisance when programming. The dog named Tiny is still a Great Dane and we would like to refer to its breed, even if we treated it as a `Pet` someplace along the line.

Fortunately, C++ does offer us a way to treat a `Dog` as a `Pet` without throwing away the name of the breed. To do this, we use pointers to dynamic object instances.

Suppose we add the following declarations:

```
Pet *ppet;
Dog *pdog;
```

If we use pointers and dynamic variables, we can treat `Tiny` as a `Pet` without losing his breed. The following is allowed:

```
pdog = new Dog;
pdog->name = "Tiny";
pdog->breed = "Great Dane";
ppet = pdog;
```

Moreover, we can still access the `breed` field of the node pointed to by `ppet`. Suppose that

```
Dog::print();
```

has been defined as follows:

```
//uses iostream
void Dog::print()
{
    cout << "name: " << name << endl;
    cout << "breed: " << breed << endl;
}
```

The statement

```
ppet->print();
```

will cause the following to be printed on the screen:

```
name: Tiny
breed: Great Dane
```

This is by virtue of the fact that `print()` is a *`virtual`* member function. (No pun intended.) We have included test code in Display 16.12.

PITFALL The Slicing Problem

Although it is legal to assign a derived class object into a base class variable, assigning a derived class object to a base class object slices off data. Any data members in the derived class object that are not also in the base class will be lost in the assignment, and any member functions that are not defined in the base class are similarly unavailable to the resulting base class object.

If we make the following declarations and assignments:

```
Dog vdog;
Pet vpet;
vdog.name = "Tiny";
vdog.breed = "Great Dane";
vpet = vdog;
```

then `vpet` cannot be a calling object for a member function introduced in `Dog`, and the data member, `Dog::breed`, is lost.

PITFALL Not Using Virtual Member Functions

In order to get the benefit of the extended type compatibility we discussed earlier, you must use *`virtual`* member functions. For example, suppose we had not used member functions in the example in Display 16.12. Suppose that in place of

```
ppet->print();
```

we had used the following:

```
cout << "name: " << ppet->name
     << " breed: " << ppet->breed << endl;
```

Display 16.12 More Inheritance with Virtual Functions (*part 1 of 2*)


```
//Program to illustrate use of a virtual function
//to defeat the slicing problem.

#include <string>
#include <iostream>
using namespace std;

class Pet
{
public:
    virtual void print();
    string name;
};

class Dog : public Pet
{
public:
    virtual void print();//keyword virtual not needed, but put
                         //here for clarity. (It is also good style!)
    string breed;
};

int main()
{
    Dog vdog;
    Pet vpet;

    vdog.name = "Tiny";
    vdog.breed = "Great Dane";
    vpet = vdog;

    //vpet.breed; is illegal since class Pet has no member named breed

    Dog *pdog;
    pdog = new Dog;
```

Display 16.12 More Inheritance with Virtual Functions (*part 2 of 2*)

```
    pdog->name = "Tiny";
    pdog->breed = "Great Dane";

    Pet *ppet;
    ppet = pdog;
    ppet->print(); // These two print the same output:
    pdog->print(); // name: Tiny breed: Great Dane

    //The following, which accesses member variables directly
    //rather than via virtual functions, would produce an error:
    //cout << "name: " << ppet->name << "  breed: "
    //     << ppet->breed << endl;
    //generates an error message: 'class Pet' has no member
    //named 'breed' .
    //See Pitfall section "Not Using Virtual Member Functions"
    //for more discussion on this.

    return 0;
}

void Dog::print()
{
    cout << "name: " << name << endl;
    cout << "breed: " << breed << endl;
}

void Pet::print()
{
    cout << "name: " << endl;//Note no breed mentioned
}
```

Sample Dialogue

```
name: Tiny
breed: Great Dane
name: Tiny
breed: Great Dane
```

This would have precipitated an error message. The reason for this is that the expression

```
*ppet
```

has its type determined by the pointer type of `ppet`. It is a pointer type for the type `Pet`, and the type `Pet` has no field named `breed`.

But `print()` was declared *`virtual`* by the base class, `Pet`. So, when the compiler sees the call

```
ppet->print();
```

it checks the *`virtual`* table for classes `Pet` and `Dog` and sees that `ppet` points to an object of type `Dog`. It therefore uses the code generated for

```
Dog::print(),
```

rather than the code for

```
Pet::print().
```

Object-oriented programming with dynamic variables is a very different way of viewing programming. This can all be bewildering at first. It will help if you keep two simple rules in mind:

1. If the domain type of the pointer `p_ancestor` is a base class for the domain type of the pointer `p_descendant`, then the following assignment of pointers is allowed:

   ```
   p_ancestor = p_descendant;
   ```

   Moreover, none of the data members or member functions of the dynamic variable being pointed to by `p_descendant` will be lost.
2. Although all the extra fields of the dynamic variable are there, you will need *`virtual`* member functions to access them.

PITFALL Attempting to Compile Class Definitions without Definitions for Every Virtual Member Function

It is wise to develop incrementally. This means code a little, then test a little, then code a little more and test a little more, and so forth. However, if you try to compile classes with *`virtual`* member functions but do not implement each member, you may run into some very-hard-to-understand error messages, even if you do not call the undefined member functions!

If any virtual member functions are not implemented before compiling, then the compilation fails with error messages similar to this: "undefined reference to *Class_Name* virtual table".
Even if there is *no derived class* and there is *only one* `virtual` member, this kind of message still occurs if that function does not have a definition.

What makes the error messages very hard to decipher is that without definitions for the functions declared `virtual`, there may be further error messages complaining about an undefined reference to default constructors, even if these constructors really are already defined.

Programming TIP
Make Destructors Virtual

It is a good policy to always make destructors virtual, but before we explain why this is a good policy we need to say a word or two about how destructors and pointers interact and about what it means for a destructor to be virtual.

Consider the following code, where `SomeClass` is a class with a destructor that is not virtual:

```
SomeClass *p = new SomeClass;
    . . .
delete p;
```

When `delete` is invoked with `p`, the destructor of the class `SomeClass` is automatically invoked. Now, let's see what happens when a destructor is marked as `virtual`.

The easiest way to describe how destructors interact with the virtual function mechanism is that destructors are treated as if all destructors had the same name (even though they do not really have the same name). For example, suppose `Derived` is a derived class of the class `Base` and suppose the destructor in the class `Base` is marked `virtual`. Now consider the following code:

```
Base *pBase = new Derived;
    . . .
delete pBase;
```

When `delete` is invoked with `pBase`, a destructor is called. Since the destructor in the class `Base` was marked `virtual` and the object pointed to is of type `Derived`, the destructor for the class `Derived` is called (and it in turn calls the destructor for the class `Base`). If the destructor in the class `Base` had not been declared as `virtual`, then only the destructor in the class `Base` would be called.

Another point to keep in mind is that when a destructor is marked as *virtual*, then all destructors of derived classes are automatically virtual (whether or not they are marked *virtual*). Again, this behavior is as if all destructors had the same name (even though they do not).

Now we are ready to explain why all destructors should be virtual. Suppose the class `Base` has a member variable `pB` of a pointer type, the constructor for the class `Base` creates a dynamic variable pointed to by `pB`, and the destructor for the class `Base` deletes the dynamic variable pointed to by `pB`. And suppose the destructor for the class `Base` is *not* marked *virtual*. Also suppose that the class `Derived` (which is derived from `Base`) has a member variable `pD` of a pointer type, the constructor for the class `Derived` creates a dynamic variable pointed to by `pD`, and the destructor for the class `Derived` deletes the dynamic variable pointed to by `pD`. Consider the following code:

```
Base *pBase = new Derived;
    . . .
delete pBase;
```

Since the destructor in the base class is not marked *virtual*, only the destructor for the class `Base` will be invoked. This will return to the freestore the memory for the dynamic variable pointed to by `pB`, but the memory for the dynamic variable pointed to by `pD` will never be returned to the freestore (until the program ends).

On the other hand, if the destructor for the base class `Base` were marked *virtual*, then when *delete* is applied to `pBase`, the destructor for the class `Derived` would be invoked (since the object pointed to is of type `Derived`). The destructor for the class `Derive` would delete the dynamic variable pointed to by `pD` and then automatically invoke the destructor for the base class `Base`, and that would delete the dynamic variable pointed to by `pB`. So, with the base class destructor marked as *virtual*, all the memory is returned to the freestore. To prepare for eventualities such as these, it is best to always mark destructors as virtual.

SELF-TEST EXERCISES

12 Why can't we assign a base class object to a derived class variable?

13 What is the problem with the (legal) assignment of a derived class object to a base class variable?

14 Suppose the base class and the derived class each have a member function with the same signature. When you have a pointer to a base class object and call a function member through the pointer, discuss what determines which function is actually called—the base class member function or the derived-class function.

CHAPTER SUMMARY

- Inheritance provides a tool for code reuse by deriving one class from another by adding features to the derived class.
- Derived class objects inherit all the members of the base class, and may add members.
- Late binding means that the decision of which version of a member function is appropriate is decided at runtime. Virtual functions are what C++ uses to achieve late binding. Polymorphism, late binding, and virtual functions are really all the same topic.
- A *protected* member in the base class is directly available to a publicly derived class's member functions.

Answers to Self-Test Exercises

1 Yes. You can plug in an object of a derived class for a parameter of the base class type. An `HourlyEmployee` is an `Employee`. A `SalariedEmployee` is an `Employee`.

2
```
class SmartBut : public Smart
{
public:
    SmartBut( );
    SmartBut(int new_a, int new_b, bool new_crazy);
    bool is_crazy( ) const;
private:
    bool crazy;
};
```

3 It is legal because `a` and `b` are marked *protected* in the base class `Smart` and so they can be accessed by name in a derived class. If `a` and `b` had instead been marked *private*, then this would be illegal.

4 The declaration for the function `get_name` is not given in the definition of `SalariedEmployee` because it is not redefined in the class `SalariedEmployee`. It is inherited unchanged from the base class `Employee`.

5
```
#include <iostream>
#include "salariedemployee.h"
using namespace std;
namespace employeessavitch
```

```
{
    class TitledEmployee : public SalariedEmployee
    {
    public:
        TitledEmployee( );
        TitledEmployee(string the_name, string the_title,
                       string the_ssn, double the_salary);
        string get_title( ) const;
        void set_title(string the_title);
        void set_name(string the_name);
    private:
        string title;
    };
}//employeessavitch
```

6

```
namespace employeessavitch
{
    TitledEmployee::TitledEmployee( )
          : SalariedEmployee( ), title("No title yet")
    {
        //deliberately empty
    }

    TitledEmployee::TitledEmployee(string the_name,
                        string the_title,
                        string the_ssn, double the_salary)
         : SalariedEmployee(the_name, the_ssn, the_salary),
                      title(the_title)
    {
        //deliberately empty
    }

    void TitledEmployee::set_name(string the_name)
    {
        Employee::set_name(title + the_name);
    }
}//employeessavitch
```

7 No. If you do not define an overloaded assignment operator or a copy constructor for a derived class, then a default assignment operator and a default copy constructor will be defined for the derived class. However, if the class

involves pointers, dynamic arrays, or other dynamic data, then it is almost certain that neither the default assignment operator nor the default copy constructor will behave as you want them to.

8 The constructors are called in the following order: first `Parent`, then `Child`, and finally `Grandchild`. The destructors are called in the reverse order: first `Grandchild`, then `Child`, and finally `Parent`.

9
```
//Uses iostream and cstdlib:
void PartFilledArray::add_value(double new_entry)
{
     if (number_used == max_number)
     {
         cout << "Adding to a full array.\n";
         exit(1);
     }
    else
    {
        a[number_used] = new_entry;
        number_used++;
    }
}

PartFilledArray::PartFilledArray
          (const PartFilledArray& object)
                : max_number(object.max_number),
                      number_used(object.number_used)
{
    a = new double[max_number];

    for (int i = 0; i < number_used; i++)
        a[i] = object.a[i];
}

void PartFilledArray::operator =
                  (const PartFilledArray& right_side)
{
    if (right_side.max_number > max_number)
    {
        delete [] a;
```

```
        max_number = right_side.max_number;
        a = new double[max_number];
    }
    number_used = right_side.number_used;

    for (int i = 0; i < number_used; i++)
        a[i] = right_side.a[i];
}

PartFilledArray::~PartFilledArray()
{
    delete [] a;
}
```

10
```
class PartFilledArrayWMax : public PartFilledArray
{
public:
   PartFilledArrayWMax(int array_size);
   PartFilledArrayWMax(const PartFilledArrayWMax& object);
   ~PartFilledArrayWMax();
   void operator= (const PartFilledArrayWMax& right_side);
   void add_value(double new_entry);
   double get_max();
private:
   double max_value;
};

PartFilledArrayWMax::PartFilledArrayWMax(int array_size)
                           : PartFilledArray(array_size)
{
    //Body intentionally empty.
    //Max_value uninitialized, since there
    //is no suitable default value.
}

/*
note the following does not work because it calls the
default constructor for PartFilledArray, but
PartFilledArray has no default constructor:
```

```
PartFilledArrayWMax::PartFilledArrayWMax(int array_size)
               : max_number(array_size), number_used(0)
{
    a = new double[max_number];
}
*/

PartFilledArrayWMax::PartFilledArrayWMax
            (const PartFilledArrayWMax& object)
                : PartFilledArray(object)
{
    if (object.number_used > 0)
    {
        max_value = a[0];

        for (int i = 1; i < number_used; i++)
            if (a[i] > max_value)
                max_value = a[i];
    }//else leave max_value uninitialized
}

//This is equivalent to the default destructor supplied
// by C++, and so this definition can be omitted.
//But, if you omit it, you must also omit the destructor
//declaration from the class definition.
PartFilledArrayWMax::~PartFilledArrayWMax()
{
    //Intentionally empty.
}

void PartFilledArrayWMax::operator =
                (const PartFilledArrayWMax& right_side)
{
    PartFilledArray::operator = (right_side);
    max_value = right_side.max_value;
}

//Uses iostream and cstdlib:
void PartFilledArrayWMax::add_value(double new_entry)
{
```

```
    if (number_used == max_number)
    {
         cout << "Adding to a full array.\n";
         exit(1);
    }

     if ((number_used == 0) || (new_entry > max_value))
         max_value = new_entry;

     a[number_used] = new_entry;
     number_used++;
}

double PartFilledArrayWMax::get_max()
{
     return max_value;
}
```

11 The output would change to

```
Discounted item is not cheaper.
```

12 There would be no member to assign to the derived class's added members.

13 Although it is legal to assign a derived class object to a base class variable, this discards the parts of the derived class object that are not members of the base class. This situation is known as the *slicing problem*.

14 If the base class function carries the `virtual` modifier, then the type of the object to which the pointer was initialized determines whose member function is called. If the base class member function does not have the `virtual` modifier, then the type of the pointer determines whose member function is called.

Programming Projects

1 Write a program that uses the class `SalariedEmployee` in Display 16.4. Your program is to define a class called `Administrator`, which is to be derived from the class `SalariedEmployee`. You are allowed to change `private` in the base class to `protected`. You are to supply the following additional data and function members:

A member variable of type `string` that contains the administrator's title, (such as Director or Vice President).

A member variable of type `string` that contains the company area of responsibility (such as Production, Accounting, or Personnel).

A member variable of type `string` that contains the name of this administrator's immediate supervisor.

A *`protected`*: member variable of type *`double`* that holds the administrator's annual salary. It is possible for you to use the existing salary member if you did the change recommended above.

A member function called `set_supervisor`, which changes the supervisor name.

A member function for reading in an administrator's data from the keyboard.

A member function called `print`, which outputs the object's data to the screen.

An overloading of the member function `print_check( )` with appropriate notations on the check.

2 Add temporary, administrative, permanent, and other classifications of employee to the hierarchy from Displays 16.1, 16.3, and 16.4. Implement and test this hierarchy. Test all member functions. A user interface with a menu would be a nice touch for your test program.

3 Give the definition of a class named `Doctor` whose objects are records for a clinic's doctors. This class will be a derived class of the class `SalariedEmployee` given in Display 16.4. A `Doctor` record has the doctor's specialty (such as "Pediatrician," "Obstetrician," "General Practitioner," etc., so use type `string`) and office visit fee (use type *`double`*). Be sure your class has a reasonable complement of constructors, accessor, and mutator member functions, an overloaded assignment operator, and a copy constructor. Write a driver program to test all your functions.

4 Create a base class called `Vehicle` that has the manufacturer's name (type `string`), number of cylinders in the engine (type *`int`*), and owner (type `Person`, given below). Then create a class called `Truck` that is derived from `Vehicle` and has additional properties: the load capacity in tons (type *`double`* since it may contain a fractional part) and towing capacity in pounds (type *`int`*). Be sure your classes have a reasonable complement of constructors, accessor, and mutator member functions, an overloaded assignment operator, and a copy constructor. Write a driver program that tests all your member functions.

The definition of the class `Person` is below. The implementation of the class is part of this Programming Project.

```
class Person
{
public:
    Person();
    Person(string the_name);
    Person(const Person& the_object);
    string get_name() const;
    Person& operator=(const Person& rt_side);
    friend istream& operator >>(istream& in_stream,
                                Person& person_object);
    friend ostream& operator <<(ostream& out_stream,
                           const Person& person_object);
private:
    string name;
};
```

5 Give the definition of two classes, `Patient` and `Billing`, whose objects are records for a clinic. `Patient` will be derived from the class `Person` given in Programming Project 4. A `Patient` record has the patient's name (inherited from the class `Person`) and primary physician, of type `Doctor` defined in Programming Project 3. A `Billing` object will contain a `Patient` object, a `Doctor` object, and an amount due of type *`double`*. Be sure your classes have a reasonable complement of constructors, accessor, and mutator member functions, an overloaded assignment operator, and a copy constructor. First write a driver program to test all your member functions, and then write a test program that creates at least two patients, at least two doctors, and at least two `Billing` records, then prints out the total income from the `Billing` records.

6 Consider a graphics system that has classes for various figures—rectangles, squares, triangles, circles, and so on. For example, a rectangle might have data members for height, width, and center point, while a square and circle might have only a center point and an edge length or radius, respectively. In a well-designed system, these would be derived from a common class, `Figure`. You are to implement such a system.

The class `Figure` is the base class. You should add only `Rectangle` and `Triangle` classes derived from `Figure`. Each class has stubs for member functions `erase` and `draw`. Each of these member functions outputs a message telling what function has been called and what the class of the calling object is. Since these are just stubs, they do nothing more than output this message. The member function `center` calls the `erase` and `draw`

functions to erase and redraw the figure at the center. Since you have only stubs for `erase` and `draw`, `center` will not do any "centering" but will call the member functions `erase` and `draw`. Also add an output message in the member function `center` that announces that `center` is being called. The member functions should take no arguments.

There are three parts to this project:

a. Write the class definitions using no virtual functions. Compile and test.
b. Make the base class member functions virtual. Compile and test.
c. Explain the difference in results.

For a real example, you would have to replace the definition of each of these member functions with code to do the actual drawing. You will be asked to do this in Programming Project 7.

Use the following `main` function for all testing:

```
//This program tests Programming Problem 6.
#include <iostream>
#include "figure.h"
#include "rectangle.h"
#include "triangle.h"
using std::cout;

int main( )
{
    Triangle tri;
    tri.draw( );
    cout <<
      "\nDerived class Triangle object calling center( ).\n";
    tri.center( ); //Calls draw and center

    Rectangle rect;
    rect.draw( );
    cout <<
      "\nDerived class Rectangle object calling center().\n";
    rect.center( ); //Calls draw and center
    return 0;
}
```

7 Flesh out Programming Problem 6. Give new definitions for the various constructors and the member functions `Figure::center`, `Figure::draw`, `Figure::erase`, `Triangle::draw`, `Triangle::erase`, `Rectangle::draw`, and `Rectangle::erase` so that the `draw` functions actually draw figures on the screen by placing the character `'*'` at suitable locations. For the `erase` functions, you can simply clear the screen (by outputting blank lines or by doing something more sophisticated). There are a lot of details in this problem, and you will have to make decisions about some of them on your own.

For additional online Programming Projects, click the CodeMate icons below.

16.8

16.9