13

Object-Oriented Programming: Polymorphism

One Ring to rule them all,
One Ring to find them,
One Ring to bring them all
and in the darkness bind
them.

—John Ronald Reuel Tolkien

The silence often of pure
innocence
Persuades when speaking
fails.

—William Shakespeare

General propositions do not
decide concrete cases.

—Oliver Wendell Holmes

A philosopher of imposing
stature doesn't think in a
vacuum. Even his most
abstract ideas are, to some
extent, conditioned by what
is or is not known in the
time when he lives.

—Alfred North Whitehead

OBJECTIVES

In this chapter you'll learn:

- How polymorphism makes programming more convenient and systems more extensible.
- The distinction between abstract and concrete classes and how to create abstract classes.
- To use runtime type information (RTTI).
- How C++ implements `virtual` functions and dynamic binding.
- How `virtual` destructors ensure that all appropriate destructors run on an object.
- How polymorphism makes programming more convenient and systems more extensible.
- The distinction between abstract and concrete classes and how to create abstract classes.

Assignment Checklist

Name: ____________________________ Date: ____________________________

Section: ____________________________

Exercises	Assigned: Circle assignments	Date Due
Prelab Activities		
Matching	YES NO	
Fill in the Blank	11, 12, 13, 14, 15, 16, 17	
Short Answer	18, 19, 20, 21	
Programming Output	22, 23	
Correct the Code	24, 25, 26, 27	
Lab Exercises		
Lab Exercise — Polymorphic Banking	YES NO	
Debugging	YES NO	
Labs Provided by Instructor		
1.		
2.		
3.		
Postlab Activities		
Coding Exercises	1, 2, 3, 4, 5	
Programming Challenges	1, 2	

Prelab Activities

Matching

Name: ____________________________ **Date:** ____________________________

Section: ____________________________

After reading Chapter 13 of *C++ How to Program, Seventh Edition*, answer the given questions. These questions are intended to test and reinforce your understanding of key concepts and may be done either before the lab or during the lab.

For each term in the column on the left, write the corresponding letter for the description that best matches it from the column on the right.

	Term	Description
___	1. `virtual` function	a) Class that is defined, but never intended to be used by the programmer to create objects.
___	2. `virtual` function table	b) Function prototypes that end with "`= 0`."
___	3. Override a `virtual` function	c) Allows objects of different classes related by inheritance to respond differently to the same message.
___	4. Dynamic binding	d) Part of C++'s run-time type information.
___	5. `virtual` base-class destructor	e) Process of replacing an inherited base-class member function with a derived-class one.
___	6. Abstract base class	f) Programming "in the general."
___	7. Pure `virtual` function	g) An executing program uses this to select the proper function implementation each time a virtual function is called.
___	8. Polymorphism	h) Occurs only off pointer or reference handles.
___	9. Concrete class	i) Ensures proper cleanup when processing dynamically allocated objects in a class hierarchy, polymorphically.
___	10. `typeid`	j) Class from which objects can be instantiated.

Prelab Activities

Name:

Fill in the Blank

Name: ___________________________ **Date:** ___________________________

Section: ___________________________

Fill in the blank for each of the following statements:

11. __________ functions allow programs to be written to process objects of types that may not exist when the program is under development.

12. __________ is implemented via `virtual` functions and dynamic binding.

13. Classes from which objects can be ___________ are called concrete classes.

14. A class is made abstract by declaring one or more __________.

15. Resolving `virtual` function references at compile-time is known as __________.

16. Objects of a(n) __________ class cannot be instantiated in a program.

17. A class with `0` pointers in the *vtable* is a(n) __________ class.

Prelab Activities

Name:

Short Answer

Name: ____________________________ **Date:** ____________________________

Section: ____________________________

In the space provided, answer each of the given questions. Your answers should be concise; aim for two or three sentences.

18. Discuss some of the problems that arise when using `switch` logic to process different objects. How do virtual functions and polymorphic programming eliminate the need for `switch` logic?

19. Briefly discuss what a *vtable* is and how it keeps track of `virtual` functions.

20. What problem arises in polymorphism when cleaning up dynamically allocated objects of a class hierarchy that has non-`virtual` destructors? How is it resolved?

Prelab Activities

Name:

Short Answer

21. What are some of the program-design advantages of using polymorphism?

Prelab Activities

Name:

Programming Output

Name: ______________________ **Date:** ______________________

Section: ______________________

For each of the given program segments, read the code and write the output in the space provided below each program. [*Note:* Do not execute these programs on a computer.]

For *Programming Output Exercises 22* and *23*, use the class definitions in Fig. L 13.1.

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

// class Oyster definition
class Oyster
{
public:
   // constructor
   Oyster( string genusString )
   {
      genus = genusString;
   } // end class Oyster constructor

   // function getPhylum definition
   string getPhylum() const
   {
      return "Mollusca";
   } // end function getPhylum

   // function getName definition
   virtual string getName() const
   {
      return "Oyster class";
   } // end function getName

   // function getGenus definition
   string getGenus() const
   {
      return genus;
   } // end function getGenus

   // print function
   virtual void print() const = 0;
private:
   string genus;
}; // end class Oyster

```

Fig. L 13.1 | `Oyster.cpp`. (Part 1 of 2.)

Prelab Activities

Name:

Programming Output

```cpp
// class VirginiaOyster definition
class VirginiaOyster : public Oyster
{
public:
   // constructor calls base-class constructor
   VirginiaOyster()
      : Oyster( "Crassostrea" )
   {
      // empty
   } // end class VirginiaOyster constructor

   // function getName definition
   virtual string getName() const
   {
      return "VirginiaOyster class";
   } // end function getName

   // print function
   virtual void print() const
   {
      cout << "Phylum: " << getPhylum()
           << "\tGenus: " << getGenus();
   } // end print function
}; // end class VirginiaOyster
```

Fig. L 13.1 | `Oyster.cpp`. (Part 2 of 2.)

22. What is output by the following program? Use class `Oyster` and `VirginiaOyster` (Fig. L 13.1).

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

#include "Oyster.cpp"

int main()
{
   VirginiaOyster oyster;
   Oyster *baseClassPtr;

   baseClassPtr = &oyster;
   baseClassPtr->print();

   cout << endl;
} // end main
```

Your answer:

Prelab Activities

Name:

Programming Output

23. What is output by the following program segment? Assume that the `Oyster` class member function `print` has been changed to that shown below.

```
// function print definition
virtual void print() const
{
   cout << "Oysters belong to Phylum " << getPhylum() << endl;
} // end function print
```

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

#include "oyster.cpp"

int main()
{
   VirginiaOyster *ptr;
   VirginiaOyster oyster;
   Oyster *oysterPtr;

   oysterPtr = &oyster;
   ptr = &oyster;

   ptr -> print();
   cout << endl;

   oysterPtr -> print();
   cout << endl << oysterPtr -> getPhylum();

   cout << endl;
} // end main
```

Your answer:

Prelab Activities

Name:

Correct the Code

Name: ______________________ **Date:** ______________________

Section: ______________________

For each of the given program segments, determine if there is an error in the code. If there is an error, specify whether it is a logic or compilation error, circle the error in the program, and write the corrected code in the space provided after each problem. If the code does not contain an error, write "no error." [*Note*: It is possible that a program segment may contain multiple errors.]

24. The following code defines an abstract class named `Base`:

```
// class Base definition
class Base
{
public:
   void print() const;
}; // end class Base
```

Your answer:

25. The following is a modified version of the definition of class `VirginiaOyster` from Fig. L 13.1. Assume member function `print` is defined in another file.

```
// class VirginiaOyster definition
class VirginiaOyster : public Oyster
{
public:
   // constructor
   virtual VirginiaOyster( string genusString )
   {
      genus = genusString;
   } // end class VirginiaOyster constructor

```

Prelab Activities

Name:

Correct the Code

```
   // constructor
   VirginiaOyster( char *genusString )
   {
      genus = genusString;
   } // end class VirginiaOyster constructor

   // print function
   void print() const;
}; // end class VirginiaOyster
```

Your answer:

Prelab Activities

Name:

Correct the Code

26. The following program defines two classes—BaseClass and DerivedClass—and instantiates an object of type BaseClass. [*Note:* Only the definitions for BaseClass and DerivedClass are shown; assume that another file is provided that contains the classes' implementations.]

```
// class BaseClass definition
class BaseClass
{
public:
   BaseClass( int = 0, int = 0 );
   virtual void display() = 0;
private:
   int x;
   int y;
}; // end class BaseClass

// class DerivedClass definition
class DerivedClass
{
public:
   DerivedClass( int = 0, int = 0, int = 0 );
   virtual void display();
private:
   int z;
}; // end class BaseClass

int main()
{
   BaseClass b( 5, 10 );
   b.display();
} // end main
```

Your answer:

Prelab Activities

Name:

Correct the Code

27. The following program segments define two classes: `Name` and `NameAndWeight`. `Name` should be an abstract base class and `NameAndWeight` should be a concrete derived class. Function `main` should declare an object of type `NameAndWeight` and print its `name` and `weight`. [*Note:* Only the definitions for `Name` and `NameAndWeight` are shown; assume files containing member function definitions have been provided elsewhere.]

```
// class Name definition
class Name
{
public:
   Name( string );
   virtual void printName() const = 0;
private:
   string name;
}; // end class Name

// class NameAndWeight definition
class NameAndWeight : public Name
{
public:
   NameAndWeight( string, int = 0 );
   virtual void displayWeight() const;
private:
   int weight;
}; // end class NameAndWeight
```

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

int main()
{
   NameAndWeight object( "name", 100 );

   cout << "name: " << object.printName() << endl;
   cout << "weight: " << object.displayWeight() << endl;
} // end main
```

Your answer:

Lab Exercises

Lab Exercise — Polymorphic Banking

Name: ____________________________ **Date:** ____________________________

Section: ____________________________

This problem is intended to be solved in a closed-lab session with a teaching assistant or instructor present. The problem is divided into five parts:

1. Lab Objectives
2. Description of the Problem
3. Sample Output
4. Program Template (Fig. L 13.2–Fig. L 13.8)
5. Problem-Solving Tips

The program template represents a complete working C++ program, with one or more key lines of code replaced with comments. Read the problem description and examine the sample output; then study the template code. Using the problem-solving tips as a guide, replace the `/* */` comments with C++ code. Compile and execute the program. Compare your output with the sample output provided. The source code for the template is available from the Companion Website for *C++ How to Program, Seventh Edition* at `www.pearsonhighered.com/deitel/`.

Lab Objectives

This lab was designed to reinforce programming concepts from Chapter 13 of *C++ How To Program, Seventh Edition*. In this lab, you will practice:

- Creating an `Account` base class that contains `virtual` functions and derived classes `SavingsAccount` and `CheckingAccount`.
- Defining `virtual` functions.
- Calling `virtual` functions.
- Downcasting with a pointer with the `dynamic_cast` operator.

Description of the Problem

Develop a polymorphic banking program using the `Account` hierarchy created in Exercise 12.10. Create a `vector` of `Account` pointers to `SavingsAccount` and `CheckingAccount` objects. For each `Account` in the `vector`, allow the user to specify an amount of money to withdraw from the `Account` using member function `debit` and an amount of money to deposit into the `Account` using member function `credit`. As you process each `Account`, determine its type. If an `Account` is a `SavingsAccount`, calculate the amount of interest owed to the `Account` using member function `calculateInterest`, then add the interest to the account balance using member function `credit`. After processing an `Account`, print the updated account balance obtained by invoking base class member function `getBalance`.

Lab Exercises

Name:

Lab Exercise — Polymorphic Banking

Sample Output

```
Account 1 balance: $25.00
Enter an amount to withdraw from Account 1: 15.00
Enter an amount to deposit into Account 1: 10.50
Adding $0.61 interest to Account 1 (a SavingsAccount)
Updated Account 1 balance: $21.11

Account 2 balance: $80.00
Enter an amount to withdraw from Account 2: 90.00
Debit amount exceeded account balance.
Enter an amount to deposit into Account 2: 45.00
$1.00 transaction fee charged.
Updated Account 2 balance: $124.00

Account 3 balance: $200.00
Enter an amount to withdraw from Account 3: 75.50
Enter an amount to deposit into Account 3: 300.00
Adding $6.37 interest to Account 3 (a SavingsAccount)
Updated Account 3 balance: $430.87

Account 4 balance: $400.00
Enter an amount to withdraw from Account 4: 56.81
$0.50 transaction fee charged.
Enter an amount to deposit into Account 4: 37.83
$0.50 transaction fee charged.
Updated Account 4 balance: $380.02
```

Template

```cpp
// Lab 1: Account.h
// Definition of Account class.
#ifndef ACCOUNT_H
#define ACCOUNT_H

class Account
{
public:
   Account( double ); // constructor initializes balance
   /* Write a function prototype for virtual function credit */
   /* Write a function prototype for virtual function debit */
   void setBalance( double ); // sets the account balance
   double getBalance(); // return the account balance
private:
   double balance; // data member that stores the balance
}; // end class Account

#endif
```

Fig. L 13.2 | Account.h.

Lab Exercises

Name:

Lab Exercise — Polymorphic Banking

```
// Lab 1: Account.cpp
// Member-function definitions for class Account.
#include <iostream>
using namespace std;

#include "Account.h" // include definition of class Account

// Account constructor initializes data member balance
Account::Account( double initialBalance )
{
   // if initialBalance is greater than or equal to 0.0, set this value
   // as the balance of the Account
   if ( initialBalance >= 0.0 )
      balance = initialBalance;
   else // otherwise, output message and set balance to 0.0
   {
      cout << "Error: Initial balance cannot be negative." << endl;
      balance = 0.0;
   } // end if...else
} // end Account constructor

// credit (add) an amount to the account balance
void Account::credit( double amount )
{
   balance = balance + amount; // add amount to balance
} // end function credit

// debit (subtract) an amount from the account balance
// return bool indicating whether money was debited
bool Account::debit( double amount )
{
   if ( amount > balance ) // debit amount exceeds balance
   {
      cout << "Debit amount exceeded account balance." << endl;
      return false;
   } // end if
   else // debit amount does not exceed balance
   {
      balance = balance - amount;
      return true;
   } // end else
} // end function debit

// set the account balance
void Account::setBalance( double newBalance )
{
   balance = newBalance;
} // end function setBalance

// return the account balance
double Account::getBalance()
{
   return balance;
} // end function getBalance
```

Fig. L 13.3 | Account.cpp.

Lab Exercises

Name:

Lab Exercise — Polymorphic Banking

```cpp
// Lab 1: SavingsAccount.h
// Definition of SavingsAccount class.
#ifndef SAVINGS_H
#define SAVINGS_H

#include "Account.h" // Account class definition

class SavingsAccount : public Account
{
public:
   // constructor initializes balance and interest rate
   SavingsAccount( double, double );

   double calculateInterest(); // determine interest owed
private:
   double interestRate; // interest rate (percentage) earned by account
}; // end class SavingsAccount

#endif
```

Fig. L 13.4 | SavingsAccount.h.

```cpp
// Lab 1: SavingsAccount.cpp
// Member-function definitions for class SavingsAccount.
#include "SavingsAccount.h" // SavingsAccount class definition

// constructor initializes balance and interest rate
SavingsAccount::SavingsAccount( double initialBalance, double rate )
   : Account( initialBalance ) // initialize base class
{
   interestRate = ( rate < 0.0 ) ? 0.0 : rate; // set interestRate
} // end SavingsAccount constructor

// return the amount of interest earned
double SavingsAccount::calculateInterest()
{
   return getBalance() * interestRate;
} // end function calculateInterest
```

Fig. L 13.5 | SavingsAccount.cpp.

```cpp
// Lab 1: CheckingAccount.h
// Definition of CheckingAccount class.
#ifndef CHECKING_H
#define CHECKING_H

#include "Account.h" // Account class definition

class CheckingAccount : public Account
{
public:
   // constructor initializes balance and transaction fee
   CheckingAccount( double, double );

```

Fig. L 13.6 | CheckingAccount.h. (Part 1 of 2.)

Lab Exercises

Name:

Lab Exercise — Polymorphic Banking

```
   /* Write a function prototype for virtual function credit,
      which will redefine the inherited credit function */
   /* Write a function prototype for virtual function debit,
      which will redefine the inherited debit function */
private:
   double transactionFee; // fee charged per transaction

   // utility function to charge fee
   void chargeFee();
}; // end class CheckingAccount

#endif
```

Fig. L 13.6 | `CheckingAccount.h`. (Part 2 of 2.)

```
// Lab 1: CheckingAccount.cpp
// Member-function definitions for class CheckingAccount.
#include <iostream>
using namespace std;

#include "CheckingAccount.h" // CheckingAccount class definition

// constructor initializes balance and transaction fee
CheckingAccount::CheckingAccount( double initialBalance, double fee )
   : Account( initialBalance ) // initialize base class
{
   transactionFee = ( fee < 0.0 ) ? 0.0 : fee; // set transaction fee
} // end CheckingAccount constructor

// credit (add) an amount to the account balance and charge fee
void CheckingAccount::credit( double amount )
{
   Account::credit( amount ); // always succeeds
   chargeFee();
} // end function credit

// debit (subtract) an amount from the account balance and charge fee
bool CheckingAccount::debit( double amount )
{
   bool success = Account::debit( amount ); // attempt to debit

   if ( success ) // if money was debited, charge fee and return true
   {
      chargeFee();
      return true;
   } // end if
   else // otherwise, do not charge fee and return false
      return false;
} // end function debit

// subtract transaction fee
void CheckingAccount::chargeFee()
{
   Account::setBalance( getBalance() - transactionFee );
```

Fig. L 13.7 | `CheckingAccount.cpp`. (Part 1 of 2.)

Lab Exercises

Name:

Lab Exercise — Polymorphic Banking

```cpp
   cout << "$" << transactionFee << " transaction fee charged." << endl;
} // end function chargeFee
```

Fig. L 13.7 | `CheckingAccount.cpp`. (Part 2 of 2.)

```cpp
// Lab 1: polymorphicBanking.cpp
// Processing Accounts polymorphically.
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

#include "Account.h" // Account class definition
#include "SavingsAccount.h" // SavingsAccount class definition
#include "CheckingAccount.h" // CheckingAccount class definition

int main()
{
   // create vector accounts
   /* Write declarations for a vector of four pointers
      to Account objects, called accounts */

   // initialize vector with Accounts
   accounts[ 0 ] = new SavingsAccount( 25.0, .03 );
   accounts[ 1 ] = new CheckingAccount( 80.0, 1.0 );
   accounts[ 2 ] = new SavingsAccount( 200.0, .015 );
   accounts[ 3 ] = new CheckingAccount( 400.0, .5 );

   cout << fixed << setprecision( 2 );

   // loop through vector, prompting user for debit and credit amounts
   for ( size_t i = 0; i < accounts.size(); i++ )
   {
      cout << "Account " << i + 1 << " balance: $"
         << /* Call the getBalance function through Account pointer i */;

      double withdrawalAmount = 0.0;
      cout << "\nEnter an amount to withdraw from Account " << i + 1
         << ": ";
      cin >> withdrawalAmount;
      /* Call the debit function through Account pointer i */

      double depositAmount = 0.0;
      cout << "Enter an amount to deposit into Account " << i + 1
         << ": ";
      cin >> depositAmount;
      /* Call the credit function through Account pointer i */

      // downcast pointer
      SavingsAccount *savingsAccountPtr =
         /* Write a dynamic_cast operation to to attempt to downcast
            Account pointer i to a SavingsAccount pointer */

```

Fig. L 13.8 | `polymorphicBanking.cpp`. (Part 1 of 2.)

Lab Exercises

Name:

Lab Exercise — Polymorphic Banking

```
      // if Account is a SavingsAccount, calculate and add interest
      if ( /* Write a test to determine if savingsAccountPtr isn't 0 */ )
      {
         double interestEarned = /* Call member function calculateInterest
                                    through savingsAccountPtr */;
         cout << "Adding $" << interestEarned << " interest to Account "
            << i + 1 << " (a SavingsAccount)" << endl;
         /* Use the credit function to credit interestEarned to
            the SavingsAccount pointed to by savingsAccountPtr*/
      } // end if

      cout << "Updated Account " << i + 1 << " balance: $"
         << /* Call the getBalance function through Account pointer i */
         << "\n\n";
   } // end for
} // end main
```

Fig. L 13.8 | polymorphicBanking.cpp. (Part 2 of 2.)

Problem-Solving Tips

1. To achieve polymorphism, declare the functions that should be called polymorphically as virtual. To indicate a virtual function within a class definition, add "virtual" before the function prototype. When the virtual functions are redefined in a derived class, those member function prototypes should also be preceded by the keyword virtual as a good programming practice.
2. To determine if a pointer to an Account object is actually pointing to a SavingsAccount object, downcast it to a SavingsAccount * using the dynamic_cast operator. If the pointer returned by this operation is not the null pointer (i.e., 0) then the object is a SavingsAccount object and that pointer can be used to access members unique to class SavingsAccount.
3. Remember that your compiler may require you to enable run-time type information (RTTI) for this particular project before this program will run correctly.

Lab Exercises

Name:

Debugging

Name: ____________________________ **Date:** ____________________________

Section: ____________________________

The program (Fig. L 13.9–Fig. L 13.15) in this section does not run properly. Fix all the compilation errors so that the program will compile successfully. Once the program compiles, compare the output with the sample output, and eliminate any logic errors that may exist. The sample output demonstrates what the program's output should be once the program's code has been corrected.

Sample Output

```
This animal is a lion
This animal's height and weight are as follows:
Height: 45      Weight: 300

Enter a new height (using standard units): 50
Enter a new weight (using standard units): 400
Here are the new height and weight values
50
400

This animal is a dog, its name is: Fido
This animal's height and weight are as follows:
Height: 60      Weight: 120

Enter a new height (using standard units): 50
Enter a new weight (using standard units): 116
Which units would you like to see the height in? (Enter 1 or 2)
        1. metric
        2. standard
2
Which units would you like to see the weight in? (Enter 1 or 2)
        1. metric
        2. standard
1
Here are the new height and weight values
50
52
```

Lab Exercises

Name:

Debugging

Broken Code

```
// Debugging: Animal.h
#ifndef ANIMAL_H
#define ANIMAL_H

#include <string>
using namespace std;

// Note: class Animal is an abstract class
// class Animal definition
class Animal
{
public:
   Animal( int = 0, int = 0 );

   void setHeight( int );
   virtual int getHeight() const = 0;

   void setWeight( int );
   virtual int getWeight() const = 0;

   virtual void print() const = 0;
private:
   int height;
   int weight;
}; // end class Animal

#endif // ANIMAL_H
```

Fig. L 13.9 | Contents of `Animal.h`.

```
// Debugging: Animal.cpp
#include <iostream>
using namespace std;

#include "Animal.h"

// default constructor
Animal::Animal( int h, int w )
{
   height = h;
   weight = w;
} // end class Animal constructor

// function print definition
virtual void Animal::print() const
{
   cout << "This animal's height and weight are as follows:\n"
        << "Height: " << height << "\tWeight: " << weight
        << endl << endl;
} // end function print

```

Fig. L 13.10 | Contents of `animal.cpp`. (Part 1 of 2.)

Lab Exercises

Name:

Debugging

```
// return height
int Animal::getHeight() const
{
   return height;
} // end function getHeight

// return weight
int Animal::getWeight() const
{
   return weight;
} // end function getWeight

// function setHeight definition
virtual void Animal::setHeight( int h )
{
   height = h;
} // end function setHeight

// function setWeight definition
virtual void Animal::setWeight( int w )
{
   weight = w;
} // end function setWeight
```

Fig. L 13.10 | Contents of `animal.cpp`. (Part 2 of 2.)

```
// Debugging: Lion.h
#ifndef LION_H
#define LION_H

#include "Animal.h"

// class Lion definition
class Lion : public Animal
{
public:
   Lion( int = 0, int = 0 );

   virtual void print() const;
}; // end class Lion

#endif // LION_H
```

Fig. L 13.11 | Contents of `Lion.h`.

```
// Debugging: Lion.cpp
#include <iostream>
using namespace std;

#include "Lion.h"

```

Fig. L 13.12 | Contents of `Lion.cpp`. (Part 1 of 2.)

Lab Exercises

Name:

Debugging

```
// default constructor
Lion::Lion( int h, int w )
   : Animal( h, w )
{
   // empty
} // end class Lion constructor

// function print definition
void Lion::print() const
{
   cout << "This animal is a lion\n";
   Animal::print();
} // end function print
```

Fig. L 13.12 | Contents of `Lion.cpp`. (Part 2 of 2.)

```
// Debugging: Dog.h
#ifndef DOG_H
#define DOG_H

#include "Animal.h"

// class Dog definition
class Dog : public Animal
{
public:
   Dog( int = 0, int = 0, string = "Toto" );

   virtual void print() const = 0;
   virtual void getHeight() const = 0;
   virtual void getWeight() const = 0;
   string getName() const;
   void setName( string );
private:
   bool useMetric( string ) const;
   string name;
   int metricHeight;
   int metricWeight;
}; // end class Dog

#endif // DOG_H
```

Fig. L 13.13 | Contents of `Dog.h`.

```
// Debugging Dog.cpp
#include <iostream>
using namespace std;

#include "Dog.h"

```

Fig. L 13.14 | Contents of `Dog.cpp`. (Part 1 of 3.)

Lab Exercises

Name:

Debugging

```
// default constructor
Dog::Dog( int h, int w, string n )
   : Animal( h, w )
{
   setName( n );
   metricHeight = h * 2.5;
   metricWeight = w / 2.2;
} // end class Dog constructor

// return name
string Dog::getName() const
{
   return name;
} // end function getName

// function setName definition
void Dog::setName( string n )
{
   name = n;
} // end function setName

// function print definition
void Dog::print() const
{
   cout << "This animal is a dog, its name is: "
        << name << endl;
   Animal::print();
} // end function print

// return height
int Dog::getHeight()
{
   if ( useMetric( "height" ) )
      return metricHeight;
   else
      return Animal::getHeight();
} // end function print

// return weight
int Dog::getWeight()
{
   if ( useMetric( "weight" ) )
      return metricWeight;
   else
      return Animal::getWeight();
} // end function getWeight

// function useMetric definition
bool Dog::useMetric( string type ) const
{
   int choice = 0;

   cout << "Which units would you like to see the "
        << type << " in? (Enter 1 or 2)\n"
        << "\t1. metric\n"
        << "\t2. standard\n";
```

Fig. L 13.14 | Contents of Dog.cpp. (Part 2 of 3.)

Lab Exercises

Name:

Debugging

```

   cin >> choice;

   if ( choice == 1 )
      return true;
   else
      return false;
} // end function useMetric
```

Fig. L 13.14 | Contents of `Dog.cpp`. (Part 3 of 3.)

```
// Debugging: Debugging.cpp
#include <iostream>
using namespace std;

#include "Animal.h"
#include "Lion.h"
#include "Dog.h"

void setHeightWeight( Animal ) const;

int main()
{
   Dog dog1( 60, 120, "Fido" );
   Lion lion1( 45, 300 );

   setHeightWeight( lion1 );
   setHeightWeight( dog1 );
} // end main

// function setHeightWeight definition
void setHeightWeight( Animal )
{
   int height;
   int weight;

   a->print();
   cout << "Enter a new height (using standard units): ";
   cin >> height;
   a->setHeight( height );

   cout << "Enter a new weight (using standard units): ";
   cin >> weight;
   a->setWeight( weight );

   height = a->getHeight();
   weight = a->getWeight();

   cout << "Here are the new height and weight values:\n"
        << height << endl
        << weight << endl << endl;
} // end function setHeightWeight
```

Fig. L 13.15 | Contents of `debugging.cpp`.

Postlab Activities

Coding Exercises

Name: ____________________________ **Date:** ____________________________

Section: ____________________________

These coding exercises reinforce the lessons learned in the lab and provide additional programming experience outside the classroom and laboratory environment. They serve as a review after you have completed the *Prelab Activities* and *Lab Exercises* successfully.

For each of the following problems, write a program or a program segment that performs the specified action:

1. Write the header file for an abstract base class named `Base`. Include a `virtual` destructor and a `virtual` `print` function.

2. Write the header file for the class `Derived` that inherits `publicly` from class `Base` that you defined in *Coding Exercise 1*. Class `Derived` has one integer as its `private` data member and should have a `print` member function.

Postlab Activities

Name:

Coding Exercises

3. Override class `Derived`'s `print` member function to print the value of the class's `private` data member.

4. Create a `Derived` object and assign its address to a `Base` pointer. Explain why this assignment is allowed by the compiler.

5. Assign the `Base` pointer from *Coding Exercise 4* to a `Derived` pointer, without using any cast operators. Explain why this assignment is not permitted by the compiler?

Postlab Activities

Name:

Programming Challenges

Name: ______________________ **Date:** ______________________

Section: ______________________

The *Programming Challenges* are more involved than the *Coding Exercises* and may require a significant amount of time to complete. Write a C++ program for each of the problems in this section. The answers to these problems are available from the Companion Website for *C++ How to Program, Seventh Edition* at `www.pearsonhighered.com/deitel/`. Pseudocode, hints and/or sample outputs are provided to aid you in your programming.

1. Modify the payroll system of Fig. 13.13–Fig. 13.23 to include `private` data member `birthDate` in class `Employee`. Use class `Date` from Fig. 11.12–Fig. 11.13 to represent an employee's birthday. Assume that payroll is processed once per month. Create a `vector` of `Employee` references to store the various employee objects. In a loop, calculate the payroll for each `Employee` (polymorphically), and add a $100.00 bonus to the person's payroll amount if the current month is the month in which the `Employee`'s birthday occurs.

Hints:

- Since all employees have a birthday, the only class that needs to be modified to add a `birthDate` is the base class `Employee`.
- Add an appropriate member function for manipulating the birthday data such as `getBirthDate`.
- Modify the `Employee` class's constructor to ensure that the `birthDate` is initialized.
- Use standard library functions of `ctime` to determine the current month based on the computer's system clock.

Postlab Activities

Name:

Programming Challenges

- Sample output:

```
Employees processed polymorphically via dynamic binding:

salaried employee: John Smith
birthday: June 15, 1944
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
birthday: December 29, 1960
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
HAPPY BIRTHDAY!
earned $770.00

commission employee: Sue Jones
birthday: September 8, 1954
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
birthday: March 2, 1965
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
old base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

deleting object of class SalariedEmployee
deleting object of class HourlyEmployee
deleting object of class CommissionEmployee
deleting object of class BasePlusCommissionEmployee
```

2. Use the `Package` inheritance hierarchy created in Exercise 12.9 to create a program that displays the address information and calculates the shipping costs for several `Packages`. The program should contain a `vector` of `Package` pointers to objects of classes `TwoDayPackage` and `OvernightPackage`. Loop through the `vector` to process the `Packages` polymorphically. For each `Package`, invoke *get* functions to obtain the address information of the sender and the recipient, then print the two addresses as they would appear on mailing labels. Also, call each `Package`'s `calculateCost` member function and print the result. Keep track of the total shipping cost for all `Packages` in the vector, and display this total when the loop terminates.

Postlab Activities

Name:

Programming Challenges

Hint:

- Sample output:

```
Package 1

Sender:
Lou Brown
1 Main St
Boston, MA 11111

Recipient:
Mary Smith
7 Elm St
New York, NY 22222

Cost: $4.25

Package 2

Sender:
Lisa Klein
5 Broadway
Somerville, MA 33333

Recipient:
Bob George
21 Pine Rd
Cambridge, MA 44444

Cost: $8.82

Package 3

Sender:
Ed Lewis
2 Oak St
Boston, MA 55555

Recipient:
Don Kelly
9 Main St
Denver, CO 66666

Cost: $11.64

Total shipping cost: $24.71
```