**CS 37 Notes Part 7: Polymorphism**

**Outline of programs:** poly1.cpp – poly3.cpp deal with animal classes to demonstrate polymorphism. Each animal has an appropriate talk method. For example, the dog says woof and the cow says moo. Poly4.cpp will be a more practical program.

Poly2.cpp will show you what happens if you do not use virtual functions and try to do polymorphism. When a request is made through a base-class pointer, the correct function in the appropriate derived-class is called.

**WHAT IS POLYMORPHISM?** Polymorphism is the ability for objects of different classes related by inheritance to respond differently to the same method call.

For example, if an object d is of type dog and the main has d.talk(), then the output is dog says woof. If an object c is of type cow and the main has c.talk(), then the output is cow says moo. Polymorphism is implemented through virtual functions.

**poly1.cpp**

Poly1 begins with a base class Mammal whose talk method outputs mammal says hi.

**class** Mammal

{

**public:**

**virtual** **void** talk() **const**

            {

                cout << "mammal says hi" << endl;

            }

};

Note the talk function in the Mammal class has the keyword virtual before the void. (Poly2 is the same as Poly1 but this keyword virtual has been removed.)

Each of the derived classes inherit from Mammal and have their own appropriate talk method. Note that the derived classes do not have the virtual keyword before their talk methods.

**class** Dog : **public** Mammal

{

**public:**

**void** talk() **const**

            {

                cout << "dog says woof" << endl;

            }

};

**class** Cat : **public** Mammal

{

**public:**

**void** talk() **const**

            {

                cout << "cat says meow" << endl;

            }

};

*You could put the virtual in and the program will work fine.* The only time the virtual needs to be in the derived talk methods is if another class is inheriting from that animal class. For example, if poodle was inheriting from dog, then the talk method in dog would have to have the virtual keyword.

The main first declares a pointer variable to a Mammal.

**int** main()

{

    Mammal\* ptr;

The pointer variable can point to any Mammal as inheritance is a “is a” relationship, eg a Dog is a Mammal. *Ptr = dog;*

The next line creates a new Cat.

    ptr = new Cat;

    cout << "for a cat ptr:   ";

    ptr->talk();

The line ptr->talk() then prints cat says meow as ptr is pointing to a Cat object.

*for a cat ptr:   cat says meow*

Next, a Bird is created and the line ptr->talk() prints bird says chirp.

*for a bird ptr:   bird says chirp*

A Dog is created next and a Mammal is created.

*for a dog ptr:   dog says woof*

With inheritance and polymorphism, a Mammal can still be created.

*for a mammal ptr:   mammal says hi*

*\*3 points: a bird is really not a mammal, bird got thrown in anyway.*

*The program should have delete ptr; when an animal is no longer needed to return the animal’s space to free memory. For example, delete ptr; should follow each line of ptr->talk(); THIS WAY THE PROGRAM USES LESS MEMORY*

**ON BINDING**

Binding means associating a data type to a variable. (ie. Int = var;) (Static means does not change data type, Dynamic means can change data type.)

This program uses dynamic binding.

|  |  |
| --- | --- |
| Static binding (means that once a variable is declared, its data type cannot be changed.) | Dynamic binding |
| We could have also written the following main which uses static binding.  // static binding main  Cat c;  cout << “for a cat: “;  c.talk();  Bird b;  cout << “for a bird: “;  b.talk();  Cow co;  cout << “for a cow: “;  co.talk();  Dog d;  cout << “for a dog: “;  d.talk();  Mammal m;  cout << “for a mammal: “;  m.talk();  return 0; | Mammal\* ptr;  ptr = new Cat;  cout << "for a cat ptr:   ";  ptr->talk();  ptr = new Bird;  cout << "for a bird ptr:   ";  ptr->talk();  ptr = new Cow;  cout << "for a cow ptr:   ";  ptr->talk();  ptr = new Dog;  cout << "for a dog ptr:   ";  ptr->talk();  ptr = new Mammal;  cout << "for a mammal ptr:   ";  ptr->talk();  return 0; |
| With static binding, when Cat c; is done, c can only be of type Cat. | With dynamic binding, the ptr can point to any type of Mammal and can be changed in the program to point to a different type of Mammal. For example, in the first main, ptr points to a Cat, then a Bird, then a Cow, then a Dog, and lastly a Mammal. |

**poly2.cpp**

Poly2.cpp only removes the virtual keyword from the Mammal talk method.

**class** Mammal

{

**public:**

**void** talk() **const***// the word virtual removed here*

            {

                cout << "mammal says hi" << endl;

            }

};

Removing the virtual stops the polymorphism from working. That is, every time the talk() method is called using static or dynamic binding, the only output we get is Mammal says hi. Without the virtual keyword, only the base methods are called.

*/\**

*for a cat ptr:   mammal says hi*

*for a bird ptr:   mammal says hi*

*for a cow ptr:   mammal says hi*

*for a dog ptr:   mammal says hi*

*for a mammal ptr:   mammal says hi*

*\*/*

**poly3.cpp**

C++ supports abstract classes and concrete classes. In this program, the Mammal class is turned into an abstract class.

**WHAT ARE ABSTRACT AND CONCRETE CLASSES?**

**abstract class** - no objects of this class can be instantiated. An abstract class is used for inheritance only.

**concrete class** – objects of this class can be instantiated. *Every class we have written up to now has been concrete as we have instantiated objects for each class.*

To make a class abstract, the class must have one or more pure virtual functions. To make/declare a pure virtual function, just set its initializer to 0.

**class** Mammal

{

**public:**

**virtual** **void** talk() **const** = 0;*// pure virtual*

};

Again, if there is one or more pure virtual functions in a class, the class is now abstract. So, for example, suppose you want to make a class abstract that has 3 methods, you can put = 0 on all 3 methods or just put = 0 on any one of the methods in the class. (??????????/)

**class** Mammal

{

**public:**

**virtual** **void** talk() **const** = 0;*// pure virtual*

**virtual** **void** wave() **const**;*// pure virtual*

};

poly3 makes the Mammal class abstract. Note that there is no code written for the talk method as no objects of type Mammal can be instantiated since talk is a pure virtual method.

|  |  |
| --- | --- |
| Before | After |
| **virtual void** talk() **const**      {        cout << "mammal says hi" << endl;      } | **virtual** **void** talk() **const** = 0; |

*The remaining methods stay the same.* Note, you cannot put = 0 in any of the derived class talk methods as the derived classes must be concrete classes since the main instantiates objects of each derived class, and they are concrete. (The base can be abstract but the derived cannot) Note that in the main, you cannot do ptr = new Mammal; as Mammal is abstract and no Mammals can be instantiated.

When working with inheritance and polymorphism, any data members or methods that are common to all the derived classes should be declared and implemented in the base class. (ie. You don’t want to repeat yourself) It is not efficient to write duplicated code where each derived class has the same method repeated. It is much better to write the repeating method once in the base class and use inheritance.

**poly4.cpp**

This program contains an Employee base class with a first name and last name data members.

**class** Employee

{

**public:**

    Employee( **const** **char** **\***, **const** **char** **\***);*// constructor*

    ~Employee();*// destructor reclaims memory*

**virtual** **float** calcPay() **const** = 0;*// pure virtual*

*//to make abstract base class*

**virtual** **void** print() **const**;

**private:**

**char** \*firstName;

**char** \*lastName;*// dynamically allocated string*

};

There are three derived classes Hourly, Crook, and Sales corresponding to three different types of Employees.

**class** Hourly: **public** Employee*//  Hourly class derived from Employee class*

{

**public:**

    Hourly(**const** **char\***, **const** **char\***, **float** = 0.0, **float**=0.0);

**virtual** **float** calcPay() **const**;

**virtual** **void** print() **const**;

**private:**

**float** wage;

**float** hours;

};

**class** Crook: **public** Employee*//  Crook class derived from Employee class*

{

**public:**

    Crook(**const** **char\***, **const** **char\***, **float** p, **float** b, **int** n, **float** v);

**virtual** **float** calcPay() **const**;

**virtual** **void** print() **const**;

**private:**

**float** pickPocket;*// amount of money for pickpocketing*

**float** bribe;*// amount of money for each bribe*

**int** numBribes;*// number of bribes*

**float** vendMachine;*// amount of money for breaking into vending machines*

};

**class** Sales: **public** Employee*//  Sales class derived from Employee class*

{

**public:**

    Sales(**const** **char\***, **const** **char\***, **float** s, **float** u, **int** q);

**virtual** **float** calcPay() **const**;

**virtual** **void** print() **const**;

**private:**

**float** salary;

**float** unitpay;*// pay for each item sold*

**int** quantity;*// number sold*

};

As all employees have a first and last name, these data members are declared in the base class rather than duplicating them in each of the three derived classes. *Each derived class has its own calcPay method as each derived class uses a different calculation to calculate the pay amount.*

**Employee is an abstract class** as **no objects of type Employee are declared in the main.** Employee’s purpose is to be inherited by each of the derived classes. Notice how employee is not in here

**int** main()

{

    cout << fixed << showpoint << setprecision(2);

    Hourly h ("Joe", "Smith",  10.5, 20.0);*// 10.5 hours at $20 per hour*

    h.print();

    cout << " earned $" << h.calcPay() << endl;

    Employee\* ptr;

    ptr = new Hourly("Joe", "Smith", 10.5, 20.0);

    ptr->print();

    cout << " earned $" << ptr->calcPay() << endl << endl;

    Sales s ("Mary", "Jones", 2000.0, 10.0, 50);*// $2000 salary*

*// sells 50 units at $10 each*

    s.print();

    cout << " earned $" << s.calcPay() << endl;

    ptr = new Sales("Mary", "Jones", 2000.0, 10.0, 50);

    ptr->print();

    cout << " earned $" << ptr->calcPay() << endl << endl;

    Crook c ("Bill", "Johnson", 100.0, 500.0, 10, 250.0);

    c.print();

    cout << " earned $" << c.calcPay() << endl;

    ptr = new Crook("Bill", "Johnson", 100.0, 500.0, 10, 250.0);

    ptr->print();

    cout << " earned $" << ptr->calcPay() << endl << endl;

    return 0;

}

The Employee class contains a constructor,

Employee::Employee( **const** **char** \*first, **const** **char** \*last)

{

    firstName = new **char**[strlen(first) + 1];

    strcpy(firstName, first);

    lastName = new **char**[strlen(last) + 1];

    strcpy(lastName, last);

}

a destructor to delete the dynamic strings when a string goes out of scope,

Employee::~Employee()

{

    delete[] firstName;

    delete[] lastName;

}

a calcPay method declaration (only, no method found)

**virtual** **float** calcPay() **const** = 0;*// pure virtual*

and a print method.

**void** Employee::print() **const** { cout << firstName << ' ' << lastName; }

Note that calcPay is a pure virtual function which makes the class abstract. *Note that there is no code written for the calcPay method in the Employee class as no pay can be calculated for an Employee as the class is abstract and no Employee objects can be instantiated. In abstract classes, you can write code for some methods that need to be shared by all the derived classes and you can have declared methods in the abstract class with no code written.*

The print method simply prints the first and last name of an employee and will be called from each of the derived classes.

**void** Hourly::print() **const**

{

    cout << "Hourly worker: " ;

    Employee::print();*// call the base-class print function*

}

The 3 derived classes are all similar. When writing programs with inheritance and polymorphism, you want to start with the base class and only one derived class. Getting the first derived class to work properly may take some effort. However, once you get one derived class to work, all the other derived classes can usually be implemented very quickly.

All three derived classes begin with their heading line which contains the fact each is inheriting from the Employee class.

**class** Hourly: **public** Employee

**class** Sales: **public** Employee

**class** Crook: **public** Employee

The constructor, calcPay method, and print method are declared followed by the data members specific to the particular derived class.

**float** Hourly::calcPay() **const**

**float** Crook::calcPay() **const**

**float** Sales::calcPay() **const**

Each constructor heading line is written with : Employee(first, last)

Hourly::Hourly (**const** **char** \*first, **const** **char** \*last,

**float** initHours, **float** initWage)

                : Employee(first, last)*// call base-class constructor*

which calls the base constructor (employee) to place the first and last names into the object. Each derived constructor then loads in the values specific to that derived class.

The calcPay method for each derived class calculates and returns the pay for that particular type of Employee. The print method prints the type of Employee and calls the base class print method to print the first and last name of the Employee.

The main instantiates each type of Employee and calls the respective print and calcPay methods based on the object type. For example, if h is of type Hourly, then h.calcPay() will call the calcPay method of the Hourly class. Both types of binding are demonstrated in the main. Again, dynamic binding is better as you can return objects to free memory when they are no longer needed and you can reuse the pointer variables. That is, ptr can point to any of the derived type objects.

**A question I usually get deals with the line Employee \*ptr in the main.** As Employee is an abstract class, how can the line Employee \*ptr work as we are not allowed to instantiate objects of an abstract class?

    Employee\* ptr;

The answer is that no Employee object is being instantiated, only a pointer variable is being declared which can then point to any of the derived class objects. For example, new Hourly or new Crook or new Sales.

(Did you notice the moral of the story for the output? The moral is crime does pay! Ha-Ha!)

**except1.cpp**

This program demonstrates exception handling. An exception means an error has occurred. It is much better for your program to handle the exception and keep processing rather than letting the compiler handle the exception. *If the compiler handles an exception, usually a garbage value will be output or the program will stop running altogether.*

This program allows the user to enter in two floats.

    cout << "enter two floats, ctrl z to quit ";

    while ( cin >> num1 >> num2)

The program then prints the two floats along with their quotient until control z is entered to stop the program.

*// the try block surrounds the area of code that may throw an exception*

*// and the code that should not execute if the exception occurs*

try

{

answer = divide(num1, num2);

      cout << "the numbers are " << setw(10) << num1 << setw(10) << num2 << endl;

cout << "the quotient is " << answer << endl << endl;

}

The exception case is if division by zero occurs. If this does occur, we want to print a message and continue processing by letting the user type in two more floats for another set of input.

catch ( DivByZeroException goof)

{

      cout << endl << "An exception has occurred: " << goof.print() << endl<< endl;

}

cout << "enter two floats, ctrl z to quit ";

Three statements are used to perform exception handling: try, catch, and throw

**float** divide( **float** a, **float** b)

{

    if (b == 0)

        throw DivByZeroException();

    return a / b;

}

For each type of exception, a class is created. The program begins with a class called DivByZeroException which declares a string called message and assigns message to “ attempt to divide by zero “

**class** DivByZeroException

{

**public:**

    DivByZeroException() : message (" attempt to divide by zero ") {}

**const** **char** **\***print() **const** {return message;}

If you had a second type of exception, then a second class would be needed with an appropriate message. For example, NotValidNumberException.

The divide function is next which uses two parameters a and b. An if statement checks if b is equal to 0 and if so, the throw statement is used to throw a DivByZeroException. (Note that what is thrown matches the name of the exception class.)

The main declares three float variables for the two numbers and the answer for the quotient and uses a while loop to keep asking the user for values until control z is entered. A try block surrounds the area of code that may throw an exception.

After the try block is the catch block which catches and handles the exception thrown in the divide function. If the user enters a value of 0 for the second input, the divide function throws the exception and the corresponding catch block is executed. The catch block in this program just prints out the error message stored in the DivByZeroException class and execution continues. If a nonzero value is entered for the second input, no exception is thrown and the catch block is skipped for that set of input and the user can input another pair or values to divide.