Introduction to Polymorphism in C++

1. Introduction:

Modern object-oriented (OO) languages provide 3 capabilities:

- encapsulation
- inheritance
- polymorphism

which can improve the design, structure and reusability of code.

Here, we'll explore how the programming capability known as polymorphism can be used in C++.

Note: You should already have some understanding of the first two concepts before attempting this material.

2. What is polymorphism?

In programming languages, polymorphism means that some code or operations or objects behave differently in different contexts.

For example, the + (plus) operator in C++:

```
4 + 5       <-- integer addition
3.14 + 2.0  <-- floating point addition
s1 + "bar"  <-- string concatenation!
```

In C++, that type of polymorphism is called overloading.

Typically, when the term polymorphism is used with C++, however, it refers to using virtual methods, which we'll discuss shortly.

3. Employee example:

Here, we will represent 2 types of employees as classes in C++:

- a generic employee (class Employee)
- a manager (class Manager)

For these employees, we'll store data, like their:

- name
- pay rate

And...we'll require some functionality, like being able to:

- initialize the employee
- get the employee's fields
• calculate the employee's pay

To help demonstrate polymorphism in C++, we'll focus on the methods that *calculate an employee's pay*.

### 4. Employee class:

Here is a class definition for a generic Employee:

```cpp
class Employee {
public:
    Employee(string theName, float thePayRate);
    string getName() const;
    float getPayRate() const;
    float pay(float hoursWorked) const;

private:
    string name;
    float payRate;
};
```

Definitions for each of the methods follow:

```cpp
Employee::Employee(string theName, float thePayRate)
{
    name = theName;
    payRate = thePayRate;
}

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

float Employee::getPayRate() const
{
    return payRate;
}

float Employee::pay(float hoursWorked) const
{
    return hoursWorked * payRate;
}
```

Note that the *payRate* is used as an hourly wage.

### 5. Manager class:

We'll also have a Manager class that is defined reusing the Employee class (i.e., via inheritance).

Remember, if a manager inherits from an employee, then it will get all the data and functionality of an employee. We can then add any new data and methods needed for a manager and *override* (i.e., *redefine*) any methods that differ for a manager.

Here is the class definition for a Manager:
#include "employee.h"

class Manager : public Employee {
public:
    Manager(string theName,
             float thePayRate,
             bool isSalaried);

    bool getSalaried() const;

    float pay(float hoursWorked) const;

protected:
    bool salaried;
};

Definitions for the additional or overridden methods follow:

Manager::Manager(string theName,
                 float thePayRate,
                 bool isSalaried)
    : Employee(theName, thePayRate)
{
    salaried = isSalaried;
}

bool Manager::getSalaried() const
{
    return salaried;
}

float Manager::pay(float hoursWorked) const
{
    if (salaried)
        return payRate;
    /* else */
    return Employee::pay(hoursWorked);
}

The pay() method is given a new definition, in which the payRate has 2 possible uses. If the manager is salaried, payRate is the fixed rate for the pay period; otherwise, it represents an hourly rate, just like it does for a regular employee.

Note: Employees paid by a salary (i.e., those that are salaried) get a fixed amount of money each pay period (e.g., week, 2 weeks, month) regardless of how many hours they work.

6. Using Employee and Manager objects

These Employee and Manager classes can be used as follows:

    #include "employee.h"
    #include "manager.h"

    ...

    // Print out name and pay (based on 40 hours work).
Employee empl("John Burke", 25.0);
cout << "Name: " << empl.getName() << endl;
cout << "Pay: " << empl.pay(40.0) << endl;

Manager mgr("Jan Kovacs", 1200.0, true);
cout << "Name: " << mgr.getName() << endl;
cout << "Pay: " << mgr.pay(40.0) << endl;
cout << "Salaried: " << mgr.getSalaried() << endl;

Recall that a Manager has all the methods inherited from Employee, like getName(), new versions for those it override, like pay(), plus ones it added, like getSalaried.

7. Why public inheritance:

Often, we want a derived class that is a "kind of" the base class:

```
Employee    <-- generic employee
    |  
Manager    <-- specific kind of employee, 
         but still an "employee"
```

In these cases, public inheritance:

```
class Manager : public Employee {
```

is the kind of inheritance that should be used.

I.e, if a Manager is truly a "kind of" Employee, then it should have all the things (i.e., the same interface) that an Employee has.

Deriving a class publicly guarantees this, as all the public data and methods from the base class remain public in the derived class.

Note: Everything that was protected in the base class remains protected in the derived class. And, those things that were private in the base class are not directly accessible in the derived class.

There is also private and protected inheritance, but they do not imply the same kind of reuse as public inheritance. With private and protected inheritance, we cannot say that the derived class is a "kind of" the base class, since the interface the base class guarantees (i.e., its public parts) becomes private or protected, respectively. Thus, private and protected inheritance represent a different way of reusing a class.

As we'll see, public inheritance makes writing generic code easier.

8. Pointer to a base class

A base class pointer can point to either an object of the base class or of any publicly-derived class:

```
Employee *emplP;
if (condition1) {
    emplP = new Employee(...);
```
This allows us, for example, to write one set of code to deal with any kind of employee:

```cpp
cout << "Name: " << emplP->getName();
cout << "Pay rate: " << emplP->getPayRate();
```

**Note:** Typically, one just needs to write *different code* only to assign the pointer to the right kind of object, but not to call methods (as above).

9. **Calling methods with base class pointers:**

As you may suspect, calling `getName()` or `getPayRate()` using an `Employee` pointer:

```cpp
cout << "Name: " << emplP->getName();
cout << "Pay rate: " << emplP->getPayRate();
```

will do the same things (return the name field or payRate field) whether the pointer points to an `Employee` or `Manager`.

That's because both classes use the exact same version of those methods--the one defined in `Employee`.

What, however, will happen when a method that was *overridden* is called?

```cpp
Employee *emplP;
if (condition1) {
    emplP = new Employee(...);
} else if (condition2) {
    emplP = new Manager(...);
}
cout << "Pay: " << emplP->pay(40.0);
```

Your first thought may be that the same thing that would happen with actual `Employee` and `Manager` objects would happen:

```cpp
Employee empl;
Manager mgr;
cout << "Pay: " << empl.pay(40.0); // calls Employee::pay()
cout << "Pay: " << mgr.pay(40.0); // calls Manager::pay()
```

In fact, that is not the case:

```cpp
Employee *emplP;
emplP = &empl; // make point to an Employee
cout << "Pay: " << emplP->pay(40.0); // calls Employee::pay()
emplP = &mgr; // make point to a Manager
cout << "Pay: " << emplP->pay(40.0); // calls Employee::pay()
```
By default, it is the type of the pointer (i.e., Employee), not the type of the object it points to (i.e., possibly Manager) that determines which version will be called:

```cpp
Employee *emplP;

if (condition1) {
    emplP = new Employee(...);
} else if (condition2) {
    emplP = new Manager(...);
}

cout << "Pay: " << emplP->pay(40.0); // calls Employee::pay()
```

We'd prefer that it call the version of `pay()` that corresponds to the type of the object pointed to:

```cpp
Employee *emplP;

emplP = &empl; // make point to an Employee
cout << "Pay: " << emplP->pay(40.0); // call Employee::pay()

emplP = &mgr; // make point to a Manager
cout << "Pay: " << emplP->pay(40.0); // please--Manager::pay()?
```

We can get that behavior by making the `pay()` method `virtual`! We do so in its declaration:

```cpp
class Employee {
public:
    ...
    virtual float pay(float hoursWorked) const;
    ...
};
```

**Note:** Once a method is declared as `virtual`, it is `virtual` in all derived classes too. We prefer to explicitly label it as `virtual` in derived classes as a reminder:

```cpp
class Manager : public Employee {
public:
    ...
    virtual float pay(float hoursWorked) const;
    ...
};
```

10. **virtual methods with references:**

The same behavior that `virtual` methods exhibit with pointers to objects extends to `references`.

For example, suppose we wanted a function to print the pay for any kind of employee. If the `pay()` method was not `virtual`, we'd have to do something like:

```cpp
typedef enum {EMPL_PLAIN, EMPL_MANAGER} KindOfEmployee;

void PrintPay0(const Employee &empl,
    KindOfEmployee kind,
    float hoursWorked)
{
    float amount;
```
switch (kind) {
    case EMPL_PLAIN:
        amount = empl.pay(hoursWorked);
        break;
    case EMPL_MANAGER:
        // convert to Manager...
        const Manager &mgr = static_cast<const Manager &>(empl);
        // ...then call its pay()
        amount = mgr.pay(hoursWorked);
        break;
}

cout << "Pay: " << amount << endl;
}

Every time a new type of employee is added, we must add another case with a nasty cast (and another value in the enumeration).

Function PrintPay0() can be used as:

    Employee empl;
    Manager mgr;

    PrintPay0(empl, EMPL_PLAIN, 40.0);
    PrintPay0(mgr, EMPL_MANAGER, 40.0);

If the pay() method is declared virtual, the function can be written much simpler:

    void PrintPay(const Employee &empl,
                  float hoursWorked)
    {
        cout << "Pay: " << empl.pay(hoursWorked) << endl;
    }

If a new class that inherits (publicly) from Employee is later added, then PrintPay() works for it too (without modification).

Function PrintPay() can be used as:

    Employee empl;
    Manager mgr;

    PrintPay(empl, 40.0);
    PrintPay(mgr, 40.0);

11. Calling virtual methods within other methods:

   The polymorphic behavior of virtual methods extends to calling them within another method.

   For example, suppose the pay() method has been declared virtual in Employee.

   And, we add a printPay() method to the Employee class:

   void printPay(const Employee &empl, float hoursWorked)
   {
       cout << "Pay: " << empl.printPay(hoursWorked) << endl;
   }
void Employee::printPay(float hoursWorked) const
{
    cout << "Pay: " << pay(hoursWorked) << endl;
}

which gets inherited in Manager without being overridden.

Which version of pay() will be called within printPay() for a Manager?

Manager mgr;
mgr.printPay(40.0);

The Manager version of pay() gets called inside of printPay() even though printPay() was only defined in Employee!

Why? Remember that:

void Employee::printPay(float hoursWorked) const
{
    ... pay(hoursWorked) ...
}

is really shorthand for:

void Employee::printPay(float hoursWorked) const
{
    ... this->pay(hoursWorked) ...
}

and we know that virtual functions behave polymorphically with pointers!

12. Design issues:

We can often write better code using polymorphism, i.e., using public inheritance, base class pointers (or references), and virtual functions.

For example, we were able to write generic code to print any employee's pay:

void PrintPay(const Employee &empl, float hoursWorked)
{
    cout << "Pay: " << empl.pay(hoursWorked) << endl;
}

That makes sense, since pay is printed the same for all employees.

The differences are only in how pay is calculated, and we were able to isolate those where they belong, in the different classes:

virtual float Employee::pay(float hoursWorked) const;
virtual float Manager::pay(float hoursWorked) const;

Nonetheless, using polymorphism to produce good designs takes thought.

For example, suppose we add a new kind of employee, a Supervisor, with one of the following two
choices of where to place the new class in the hierarchy:

```
         a) Employee       b) Employee
              |            /    \
         Manager   Manager  Supervisor
              |               |
                 Supervisor
```

If we later write a function that takes a Manager reference:

```cpp
void GiveRaise(Manager &mgr);
```

*Which class hierarchy would allow us to pass a Supervisor to GiveRaise()*?

13. **Exercise:**

Take the code we've provided for the Employee class (employee.h and employee.cpp) and the Manager class (manager.h and manager.cpp).

1. Add a method print() to the Employee class that prints out some employee data for a pay period. E.g.:

   ```
   Name: John Burke
   Pay rate: 25
   Pay: 1000
   ```

   Your print() method should print out the name and pay rate itself, but it should call the printPay() method to print the pay.

   **Note:** print() will have to receive the number of hours worked.

   Your code should compile and run correctly with the test program polytest.cpp.

2. After getting the first part to work, consider that we want the pay for a Manager to be printed differently than for an Employee. I.e., print the manager's pay labelled as "Salary:" if they are salaried, and as "Wages:" if they are not.

   Do so by overriding (i.e., redefining) the printPay() method in Manager.

   Make any other changes that are necessary for other methods, like print(), to work correctly with printPay() (*Hint: use the virtual mechanism*).

   **Constraint for this exercise:** Only make methods virtual if it is necessary to make the above work.