Lecture 34 — Polymorphism & Inheritance

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

August 27, 2016

Acknowledgments: W.D. Bishop

Object-Oriented Design Revisited

So far, our discussion of object-oriented design has been limited to relatively simple classes.

Often, object-oriented design provides little benefit for “toy” applications such as the applications developed in this course.

Many of the things we have written in the class can be done with one class, or a handful of classes.

For large application programs developed by a team of software developers, object-oriented design is necessary.

  Programs will be too complex to leave unmanaged.

Coping with Evolving Software Demands

Two key characteristics are often desirable to cope with the evolving demands of software:

1. Polymorphism
   Define methods and collections that work with a variety of types
2. Extensibility
   Enhance existing types without changes to already-existing code

Not all object-oriented languages provide full support for both polymorphism and extensibility.

Polymorphism

The word polymorphism comes from Greek and means “many forms”.

Polymorphic code works for or on multiple types of object.
 The code is “generic” or flexible in what it accepts.

The same function, method, or collection can be used with many different types (without relying on type promotion).

Examples of Polymorphism

We have already seen some examples of polymorphism in the course.

`vector` is a polymorphic collection:
We can put many different types of object in the collection.

`std::sort` is a polymorphic function:
It will sort the objects in the vector, whatever kind of object they are.

Polymorphism exploits characteristics common to a set of types.
Example: if two objects can be compared, they can be sorted.
… regardless of the actual types of the instances.

Polymorphic Functions

A polymorphic function operates on a related set of object types.

`sort` does not need to know the type of the objects, but it does need to know how to compare the objects in the array.

Type Conversion

We have already done type conversion with explicit casting.
  Example: `double d = (double) x;`

Polymorphic functions and collections often benefit from the use of run-time type conversion.

For example, arrays of objects can be created. At run-time:
  The type of array elements can be identified.
  Entries can be converted to a compatible type for processing.

Inheritance

Inheritance: a class is based upon (or derived from) another class.

The original class is referred to as the “base” or “parent” class.
We say the derived class “inherits” from the parent class.

What does it inherit? Fields, methods, et cetera.

Inheritance provides extensibility and reduces duplicate code.

This provides two advantages:

1. Reduce the amount of code required for objects that are related
2. Promote the consistent behaviour of related objects

C++ Inheritance Rules

In C++, a class may inherit from multiple parent classes.

This has the potential to get confusing, but is very flexible.

Some languages like C# only allow one parent class.

Inheriting from a Base Class

The syntax for inheriting from a base class uses a colon (`:`).

attributes modifiers class *identifier* : *base*

Example: `class DerivedClass :  BaseClass`

Followed by the internal implementation of the derived class.

Note that the implementation of the derived class does not repeat the member function and variable declarations of the base class.

Unless there is a need for them to have a different implementation.

Inheritance Syntax

```
class BaseClass
{
    // Declares a base class from which other classes
    // may be derived.  This class will automatically
    // inherit from the object class.
};

class DerivedClass : BaseClass
{
    // Declares a derived class that inherits fields
    // and functions from the BaseClass.
};
```

Employee Example

Let’s start out with a simple class Employee:

```
class Employee
{
  protected:
    string name;
    string title;
    string sin;

  public:
     Employee( string n, string t , string s);
};
```

Employee Example

Now let's add a new kind of `Employee`: the `HourlyEmployee`.

```
class HourlyEmployee : Employee
{
  private:
    double wage;
    double weeklyHours;

  public:
    HourlyEmployee( string n, string t, string s,
                           double w, double h );
};
```

Observations on `HourlyEmployee`

The `HourlyEmployee` class inherits all of the fields and functions of the `Employee` class. They need not be duplicated.

It also adds two new fields only applicable to hourly employees.

Another new keyword appeared on that slide: `protected`
 Let's take a look at that on the next slide.

Access Modification

In addition to the access modifier keywords `public` and `private`, there is a third option: `protected`.

Declaring a field or method as `protected` means it is accessible only from within the class where it is declared and <u>any derived class</u>.

In some cases, you may not want a derived class to be able to access member fields directly; in that case, use `private`.

Employee Example

Now let's add another kind of `Employee`: the `SalaryEmployee`.

```
class SalaryEmployee : Employee
{
  protected:
    double salary;

  public:
    SalaryEmployee( string n, string t,
                           string s, double m );
}
```

Employee Example

We can use the `:` syntax to reduce the amount of code we have to write here by calling the constructor of the base class:

```
class SalaryEmployee : Employee
{
    double salary;

    public SalaryEmployee( string n, string t,
                           string s, double m )
                           : Employee (n, t, s);
}
```

Observations on `SalaryEmployee`

The `SalaryEmployee` class is closely related to `HourlyEmployee`.

Both classes inherit from the `Employee` class.
  Common elements need only be defined once.

Inheritance represents the "is-a" relationship:
  Hourly Employee is an Employee;
  Salary Employee is an Employee.

Because of this relationship, this is legal:

```
Employee* e = new HourlyEmployee( );
```

We can refer to an instance of `HourlyEmployee` as if it's an instance of `Employee`: an hourly employee is an employee.

Moving Beyond Salary and Hourly

Now suppose you wanted to add a category of employee: `Manager`.

You may decide that `Manager` is a kind of `SalaryEmployee`.

A `Manager` has a list of employees who report to him or her.
Managers can also get bonuses at the end of the year.

Solution: extend `SalaryEmployee` to make `Manager`.

The Manager

```
class Manager : SalaryEmployee
{
  protected:
    vector reports; // Remember to initialize!
    double bonus;

    public:
     Manager( string n, string t,  string s, double m,
                     double b );
}
```

More About the `Manager`

`Manager` inherits from `SalaryEmployee` which in turn inherits from `Employee`.

The “is-a” relationship is transitive:
Manager is a Salary Employee;
Salary Employee is an Employee;
Therefore Manager is an Employee.

Because of this relationship, this is also legal:

```
Employee* e = new Manager( );
```

Overriding Methods

Overriding a method is the term for replacing the behaviour of a base class by writing a new implementation of that method.

Suppose we add the following method to `SalaryEmployee`:

```
double SalaryEmployee::ComputePay( )
{
    return salary;
}
```

This method is available and can be called on instances of `SalaryEmployee` and `Manager`.

Overriding Methods

Now imagine we'd like to override this behaviour in `Manager` to account for the fact that managers can get a bonus.

```
double Manager::ComputePay( )
{
    return salary + bonus;
}
```

`ComputePay( )` can still be called on `Manager` and `SalaryEmployee`.

Overriding Methods

When calling `ComputePay( )` on an instance of `SalaryEmployee`, the method in `SalaryEmployee` is run.

When calling `ComputePay( )` on an instance of `Manager`, the method in `Manager` is run.

  Why? It is overidden in `Manager`.

What if the code just refers to an instance of `SalaryEmployee` and we're not sure if it's a manager or not?

Inheritance and Polymorphism

At run-time, the system knows what kind of object a given instance is.

Thus, the code may refer to `Employee`, but the system knows if it is really a `HourlyEmployee`, `SalaryEmployee`, or `Manager`.

It will therefore execute the appropriate method based on the actual type of the instance.

Inheritance Plus Polymorphic Functions

Earlier we asserted that a function can be polymorphic without having a generic parameter. Inheritance is the reason for this.

Suppose we have a method `double ComputeArea( Shape s )` .

Let's say that `Shape` is a base class and there are many derived classes: `Triangle, Rectangle, Circle, Polygon`...

When calling `ComputeArea`, because `Triangle` derives from `Shape` it is legal to provide a `Triangle` instance as the actual parameter.