8: POLYMORPHISM

Programming Technique II

(SECJ1023)

9.1: Introduction to Polymorphism

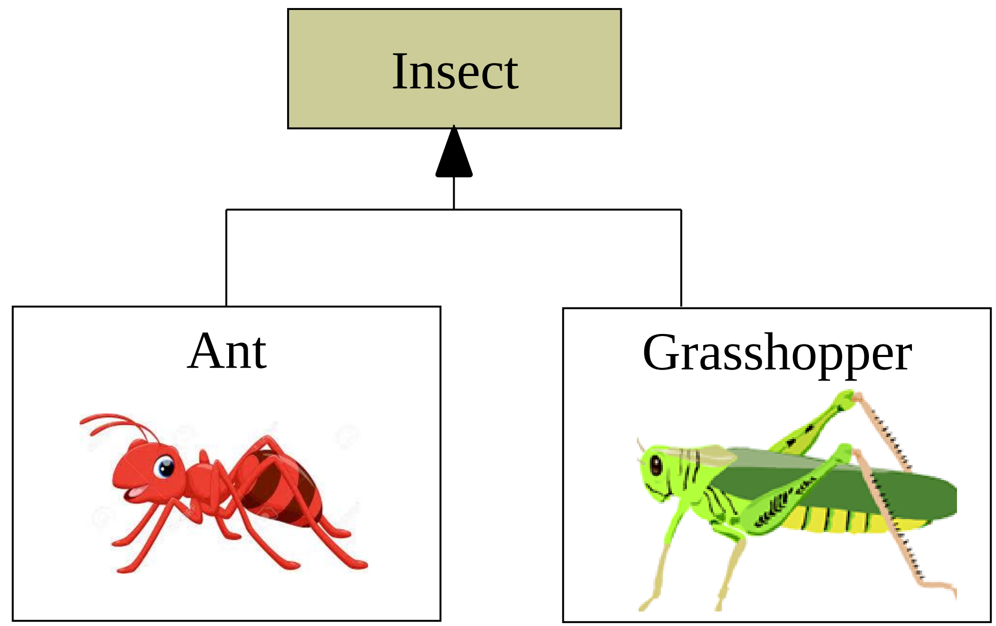
What is Polymorphism?

- **Polymorphism** is the ability of objects to **perform the same actions differently.**

Example 1:

Insects have the ability to **<u>move</u>** from one point to another.

However, the way they perform their **movement are different**



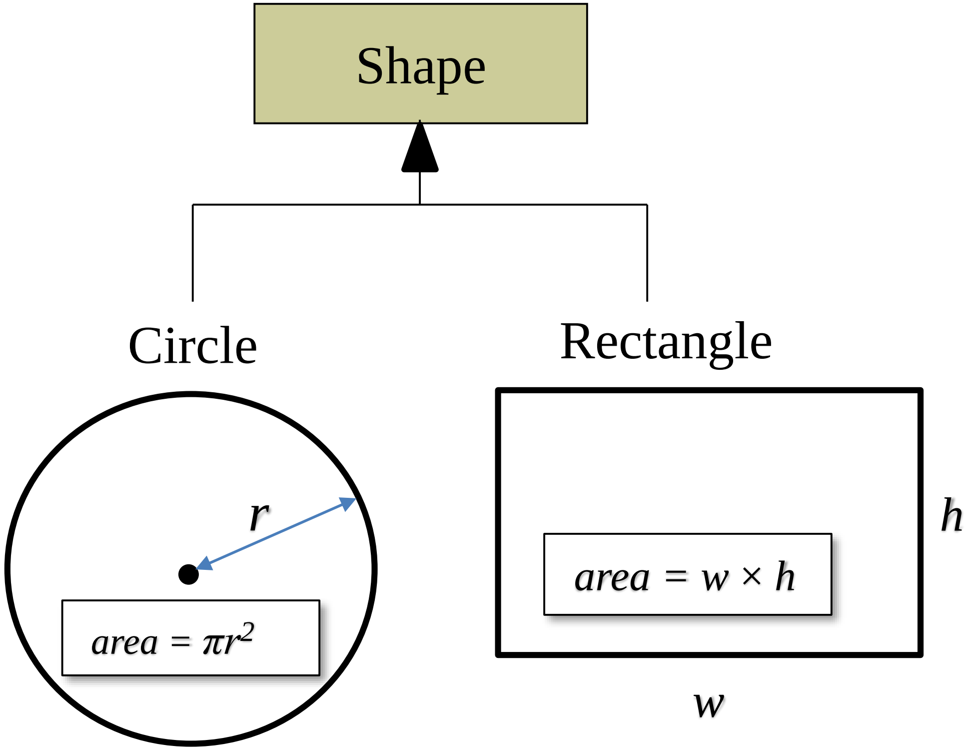
Ants **<u>move</u>** by **crawling**

Grasshoppers **<u>move</u>** by **jumping**

What is Polymorphism?

Example 2:

All geometrical shapes have the characteristic regarding area. However, calculating the area is different from one shape to another.



What is Polymorphism?

- **Same actions** but **different behaviors**? What does this mean in programming context?

- It is the same action because the classes use the same name (including parameters) for the methods.

- It is different behavior because the code in the method definition is different one to another.

What is Polymorphism?

Same actions but different behaviors

Example:

```
class Circle : public Shape{
    private:
        double r;

    public:

        double getArea(){

            return PI*r*r;
        }

};
```

```
class Rectangle : public Shape{
    private:
        double width,length;

    public:

        double getArea(){

            return width*length;
        }

};
```

Same actions: `Circle` and `Rectangle` use the same name for the method calculating the area, i.e., `getArea()`

What is Polymorphism?

Same actions but different behaviors

Example:

```
class Circle : public Shape{
    private:
        double r;

    public

        double getArea(){

            return PI*r*r;
        }

};
```

```
class Rectangle : public Shape{
    private:
        double width,length;

    public

        double getArea(){

            return width*length;
        }

};
```

Different behaviors: the code implementing the calculation of area inside each of the methods `getArea` is different, for the classes `Circle` and `Rectangle`.

9.2: Polymorphisms and Virtual Member Functions (Methods)

Terminology

- **Overloaded methods**: two or more methods share the same name but parameters are different. (either one of below)
  - Different data types of parameters
  - Different number of parameters

- **Redefined methods**: child or derived classes define methods with exactly the same name and parameters as used in the parent or base class.

- **Overridden methods**: it is similar to redefined methods except they are dynamically bound.

Terminology

Overloaded vs Redefined vs Overridden Methods

Example:

Overloaded methods: **set**

Redefined method: **print**

Overridden method: **whoAmI**

```
class Person{
    protected:
        string name;
    public:
        Person(string n="") : name(n){}

        void set(string n) {name=n;}

        void set() { cout << "Enter the name => "; cin >> name; }

        void print() const{cout << "Person's Name: " << name << endl;}

        virtual void whoAmI() const{cout << "I am a Person" << endl; }

};

class Student : public Person{
    private:
        string matric;
    public:
        Student(string m="",
                string n="") : Person(n), matric(m) {}

        void print() const{
            cout << "Student's Name   : " << name << endl;
            cout << "Student's Matric: " << matric << endl;
        }

        void whoAmI() const{ cout << "I am a Student" << endl;   }
};
```

Virtual Member Functions (Methods)

- Virtual methods are methods that are dynamically bound. i.e. the binding is decided at runtime.

- At runtime, C++ decides to which version of the method should be bound by looking at the type of object making the call.

- Virtual methods are defined with the keyword `virtual`:

  Example

  ```
  virtual void print()const {...}
  ```

- Methods defined without the keyword `virtual` are called non-virtual methods and C++ uses static binding for them, i.e., the binding is decided at compile time.

Static vs Dynamic Binding

- The method name in an object is used for reference only.

- The method needs to be bound to code (i.e. the definition) in order to perform its actions.

- Method binding can occur in two situations:
  - Pre-bound, before the program runs => **static binding**
  - During the execution of program => **dynamic binding**

Static vs Dynamic Binding

Static Binding

- Static binding is about binding "at compile time".
  - The code which the method to be bound with is decided by the compiler.

- Methods that are statically bound are constant or unchanged throughout the execution of programs. Thus term static binding.

- All methods (except overridden methods) are statically bound (e.g., redefined, overloaded, constructors, destructors, etc.)

Static vs Dynamic Binding

Dynamic Binding

- Dynamic binding is about binding "at runtime".
  - The code which the method to be bound with is decided by the program at runtime (not by the compiler).

- Methods that are dynamically bound can change from one code to another. Thus term dynamic binding.

- Dynamic binding applies to overridden methods.
  - This is done by specifying the methods as `virtual` (in the parent class)

Static vs Dynamic Binding

Problems with Static Binding

```cpp
#define PI 3.14

class Shape{
    public:
        Shape(){}
        int getArea() const { return 0;}
};

class Circle : public Shape{
    private:
        double r;
    public:
        Circle(int _r){r=_r;;}

        int getArea() const { return PI*r*r;}
};

class Rectangle : public Shape{
    private:
        int width, length;
    public:
        Rectangle(int w, int l){width = w; length = l;}

        int getArea() const { return width * length;}
};

```

```cpp
int main()
{
    Shape *p;

    Shape s;
    Circle c(10);
    Rectangle r(2,6);

    p = &s;
    cout << "Shape area = "  << p->getArea() << endl;

    p = &c;
    cout << "Circle area = " << p->getArea() << endl;

    p = &r;
    cout << "Rectangle area = " << p->getArea() << endl;

    return 0;
}
```

Program output:

```
Shape area = 0
Circle area = 0
Rectangle area = 0
```

Output error for the area of circle and rectangle

Static vs Dynamic Binding

Problems with Static Binding

- The method **`getArea`** associated with `p` is pre-bound at compile time (i.e. static binding) and remain unchanged at runtime.

- Thus, every time calling to the method `getArea` from `p` (lines 40, 43 and 46) always invoke the `getArea` of class `Shape` (because `p` is of type `Shape` pointer). As a result all the output for the area are 0.

- This problem can be solved using dynamic binding.

Static vs Dynamic Binding

The Solution with Dynamic Binding

```
#define PI 3.14

class Shape{
    public:
        Shape(){}
        virtual int getArea() const { return 0;}
};

class Circle : public Shape{
    private:
        double r;
    public:
        Circle(int _r){r=_r;}

        int getArea() const { return PI*r*r;}
};

class Rectangle : public Shape{
    private:
        int width, length;
    public:
        Rectangle(int w, int l){width = w; length = l;}

        int getArea() const { return width * length;}
};
```

Specify as `virtual` to use dynamic binding

`vrtual` is only placed in the parent class. The method becomes virtual in all child classes automatically.

```
int main()
{
    Shape *p;

    Shape s;
    Circle c(10);
    Rectangle r(2,6);

    p = &s;
    cout << "Shape area = "  << p->getArea() << endl;

    p = &c;
    cout << "Circle area = " << p->getArea() << endl;

    p = &r;
    cout << "Rectangle area = " << p->getArea() << endl;

    return 0;
}
```

Program output:

```
Shape area = 0
Circle area = 314
Rectangle area = 12
```

Static vs Dynamic Binding

The Solution with Dynamic Binding

- The method **`getArea`** associated with `p` is now bound at runtime (not pre-decided anymore by the compiler)

- At runtime, C++ determines the type of object making the call, and binds the method to the appropriate version of the function.
  - At line 42: `p` is pointing to a `Circle` object, `c.` Thus, the method `getArea` associated with `p` is now bound to the `getArea` of the class `Circle`
  - At line 45: `p` is pointing to a `Rectangle` object, `r.` Thus, the method `getArea` associated with `p` is now bound to the `getArea` of the class `Rectangle`

Requirements for the Implementation of Polymorphism

1. The method must be able to be dynamically bound
   - This is done by making it as a virtual method.
   - Place the keyword `virtual` only in the parent class.

Place `virtual` for the method to be made polymorphic. This will allow the method to be dynamically bound

```
#define PI 3.14

class Shape{
    public:
        Shape(){}
        virtual int getArea() const { return 0;}
};

class Circle : public Shape{
    private:
        double r;
    public:
        Circle(int _r){r=_r;;}

        int getArea() const { return PI*r*r;}
};

class Rectangle : public Shape{
    private:
        int width, length;
    public:
        Rectangle(int w, int l){width = w; length = l;}

        int getArea() const { return width * length;}
};
```

Requirements for the Implementation of Polymorphism

2. The child classes need to override the polymorphic methods.

```
#define PI 3.14

class Shape{
    public:
        Shape(){}
        virtual int getArea() const { return 0;}
};

class Circle : public Shape{
    private:
        double r;
    public:
        Circle(int _r){r=_r;;}

        int getArea() const { return PI*r*r;}
};

class Rectangle : public Shape{
    private:
        int width, length;
    public:
        Rectangle(int w, int l){width = w; length = l;}

        int getArea() const { return width * length;}
};
```

The class Circle overrides the method `getArea`

The class Rectangle overrides the method `getArea`

Requirements for the Implementation of Polymorphism

3. Must use parent class pointers.

```
void displayArea(const Shape *p)
{
    cout << "Area = "  << p->getArea() << endl;
}

int main()
{
    Shape s;
    Circle c(10);
    Rectangle r(2,6);

    displayArea(&s);

    displayArea(&c);

    displayArea(&r);


    return 0;
}
```

The pointer must be of type parent class.

Parameter `p` can accept any address of object of `Shape`, `Circle`, or `Rectangle`, because it is of type `Shape` pointer.

Any object of child classes or the parent class can fit here. A circle or rectangle is a kind of shape.

Program output:

```
Area = 0
Area = 314
Area = 12
```

Requirements for the Implementation of Polymorphism

- If you want to use a list, you also need to use pointers, i.e. arrays of parent class pointers.
  - Note that each element of the array is a pointer not an object.

Example:

Note: for the program below, the function `displayArea` *from previous program is excluded*

```
int main()
{
    Shape s;
    Circle c(10);
    Rectangle r(2,6);

    Shape *list[3] = { &s, &c, &r };

    for (int i=0; i<3; i++)
        cout << "Area = "
             << list[i]->getArea()
             << endl;


    return 0;
}
```

Each element of the array is a pointer of shape. Circle, `c` and rectangle, `r` can fit here because they are also type of shape.

The list is an array of parent class pointers.

We use arrow operator here because each element of the array is a pointer.

Program output:

```
Area = 0
Area = 314
Area = 12
```

Parent Class Pointers

- The pointer must be declared as of parent class pointer so that any object of the child classes and the parent class can fit. In other words to make it more general.

Example:

```
void displayArea(const Shape *p)
{
    cout << "Area = "  << p->getArea() << endl;
}
```

We can pass the address of any object of class `Circle (c)`, `Rectangle` (r) and `Shape` (s) to the function `displayArea`, because objects `c`, `r` and `s` are all of type `Shape`.

```
Shape s;
Circle c(10);
Rectangle r(2,6);

displayArea(&s);

displayArea(&c);

displayArea(&r);
```

Parent Class Pointers

- However, the parent class pointer only knows about public members of the parent class.
- The pointer cannot refer to any member in the child classes

Example:

```
class Shape{
    public:
        Shape(){}
        virtual int getArea() const { return 0;}
};

class Circle : public Shape{
    private:
        double r;
    public:
        Circle(int _r){r=_r;;}

        int getArea() const { return PI*r*r;}

        void printRadius() const{
            cout << "Radius: "
                 << r << endl;
        }
};
```

```
int main()
{
    Shape *p;
    Circle c(10);

    p = &c;

    cout << "Area = "
         << p->getArea()
         << endl;

    p->printRadius();

    return 0;
}
```

The pointer `p` is pointing to a `Circle` object, `c`

This is **fine** because class `Shape` has the method `getArea`

This produces an **error** because class `Shape` does not have the method `printRadius`. Instead, this method belongs the child class, `Circle`

Virtual Destructors

- By default destructors are bound statically (i.e., non-virtual)

- If a class could ever become a parent class, it is better to specify its destructor virtual to allow dynamic binding.

Example:

The following program shows the effect with **non-virtual destructors**.

```
5  class Parent{
6      protected:
7          string name;
8      public:
9          Parent(string n=""){
10             name = n;
11         }
12
13         ~Parent(){
14             cout << "Destroy Parent object"
15                  << endl;
16         }
17 };
18
19 class Child : public Parent{
20     private:
21         int age;
22     public:
23         Child(int a=0, string n="") : Parent(n){
24             age = a;
25         }
26
27         ~Child(){
28             cout << "Destroy Child object"
29                  << endl;
30         }
31 };
```

```
33 int main()
34 {
35     Parent *ptr;
36
37     ptr = new Child(20, "Ali");
38
39     delete ptr;
40
41     return 0;
42 }
```

Program output:

```
Destroy Parent object
```

Although `ptr` is pointing to a `Child` object (created at Line 37), deleting the object (Line 39) will call to the `Parent`'s destructor because the compiler performs static binding to the destructor.

Virtual Destructors

Example:

The previous program is modified by making the destructor **virtual**

```
5   class Parent{
6       protected:
7           string name;
8       public:
9           Parent(string n=""){
10              name = n;
11          }
12
13          virtual ~Parent(){
14              cout << "Destroy Parent object"
15                   << endl;
16          }
17  };
18
19  class Child : public Parent{
20      private:
21          int age;
22      public:
23          Child(int a=0, string n="") : Parent(n){
24              age = a;
25          }
26
27          ~Child(){
28             cout << "Destroy Child object"
29                  << endl;
30          }
31  };
```

```
33  int main()
34  {
35      Parent *ptr;
36
37      ptr = new Child(20, "Ali");
38
39      delete ptr;
40
41      return 0;
42  }
```

Program output:

```
Destroy Child object
Destroy Parent object
```

Now, deleting the object (Line 39) will call to the `Child`'s destructor because the destructor is specified `virtual` (Line 13). This allows C++ to perform dynamic binding to the destructor.

9.3: Abstract Base Classes and Pure Virtual Functions

Pure Virtual Methods and Abstract Parent Classes

- A pure virtual method is a method in a parent class declared as `virtual` but without any definition.

- Child classes must override pure virtual methods.

- A pure virtual method is indicated by the = 0 as shown below:

```
virtual void methodName() = 0;
```

Pure Virtual Methods and Abstract Parent Classes

- An abstract class is a class which cannot have any objects.
  - We cannot create an instance from this class.
  - It serves as a basis for child classes that may have objects.

- An abstract class is created when a class contains one or more pure virtual methods.

Example:
Turning the class `Shape` to be an abstract class by making the method `getArea` **pure virtual**

```
class Shape{
    public:
        virtual int getArea() const = 0;
};
```

Now, we cannot create any object from the class `Shape`.
Thus the following code would result in an error.

```
Shape s;
```