Polymorphism and Derived Classes

Chapter 11

Basic Programming 2

Outline

- Polymorphism and Derived Classes
  - Implemented with Virtual Methods
  - Slicing
  - Virtual Copy Constructors
- Making Use of Advanced Polymorphism
  - Problems with Single Inheritance
  - Abstract Data Types
  - Implementing Pure Virtual Functions
  - Complex Hierarchies of Abstraction

Dasar Teori

- Polymorphism sesuai dengan asal-usul kata pembentuknya berarti “mempunyai banyak bentuk”. Dalam wujudnya, polymorphism dapat beroperasi pada dua aras, yaitu saat kompilasi dan saat eksekusi.
- Overloading terhadap fungsi dan operator merupakan bentuk polymorphism saat kompilasi.
- *Late binding* atau *dynamic binding* merupakan bentuk polymorphism saat eksekusi.

Dasar Teori

- Untuk mendeklarasikan elemen sebuah kelas yang akan kita definisi ulang di kelas turunan maka harus didahului dengan keyword **virtual** maka pointer ke obyek yang menunjuk klas tersebut dapat digunakan dengan baik.

1. Implemented with Virtual Methods

Mammal8.cpp

```cpp
#include <iostream>

class Mammal
{
        public:
                Mammal():age(1) { std::cout << “Mammal constructor ...\n”; }
                ~Mammal() { std::cout << “Mammal destructor ...\n”; }
                void move() const { std::cout << “Mammal, move one step\n”; }
                virtual void speak() const { std::cout << “Mammal speak!\n”; }

        protected:
                int age;
};

class Dog : public Mammal
{
        public:
                Dog() { std::cout << “Dog constructor ...\n”; }
                ~Dog() { std::cout << “Dog destructor ..\n”; }
                void wagTail() { std::cout << “Wagging tail ...\n”; }
                void speak() const { std::cout << “Woof!\n”; }
                void move() const { std::cout << “Dog moves 5 steps ...\n”; }
};

int main()
{
        Mammal *pDog = new Dog;
        pDog->move();
        pDog->speak();
        return 0;
}
```

Mammal9.cpp

```
#include <iostream>

class Mammal
{
        public:
                Mammal():age(1) { }
                ~Mammal() { }
                virtual void speak() const { std::cout << “Mammal speak!\n”; }
                protected:
                int age;
};

class Dog : public Mammal
{
        public:
                void speak() const { std::cout << “Woof!\n”; }
};

class Cat : public Mammal
{
        public:
                 void speak() const { std::cout << “Meow!\n”; }
};

class Horse : public Mammal
{
        public:
                void speak() const { std::cout << “Whinny!\n”; }
};

class Pig : public Mammal
{
        public:
                void speak() const { std::cout << “Oink!\n”; }
};

```

...

```
int main()
{
        Mammal* array[5];
        Mammal* ptr;
        int choice, i;
        for (i = 0; i < 5; i++)
        {
                std::cout << “(1) dog (2) cat (3) horse (4) pig: “;
                std::cin >> choice;
                switch (choice)
                {
                        case 1:
                        ptr = new Dog;
                        break;
                        case 2:
                        ptr = new Cat;
                        break;
                        case 3:
                        ptr = new Horse;
                        break;
                        case 4:
                        ptr = new Pig;
                        break;
                        default:
                        ptr = new Mammal;
                        break;
                }
                array[i] = ptr;
        }
        for (i=0; i < 5; i++)
        {
                array[i]->speak();
        }
        return 0;
}
```

2. Slicing

Mammal10.cpp

```
#include <iostream>

class Mammal
{
        public:
                Mammal():age(1) { }
                ~Mammal() { }
                virtual void speak() const { std::cout << “Mammal speak!\n”; }
        protected:
                int age;
};

class Dog : public Mammal
{
        public:
                void speak() const { std::cout << “Woof!\n”; }
};

class Cat : public Mammal
{
        public:
                void speak()const { std::cout << “Meow!\n”; }
};

void valueFunction(Mammal);
void ptrFunction(Mammal*);
void refFunction(Mammal&);

```

…

```
int main()
{
        Mammal* ptr=0;
        int choice;
        while (1)
        {
                bool fQuit = false;
                std::cout << “(1) dog (2) cat (0) quit: “;
                std::cin >> choice;
                switch (choice)
                {
                case 0:
                fQuit = true;
                break;
                case 1:
                ptr = new Dog;
                break;
                case 2:
                ptr = new Cat;
                break;
                default:
                ptr = new Mammal;
                break;
        }
        if (fQuit)
        {
                break;
        }
        ptrFunction(ptr);
        refFunction(*ptr);
        valueFunction(*ptr);
        }
        return 0;
}

```

…

```
void valueFunction(Mammal mammalValue) // This function is called last
{
        mammalValue.speak();
}

void ptrFunction (Mammal *pMammal)
{
        pMammal->speak();
}

void refFunction (Mammal &rMammal)
{
        rMammal.speak();
}
```

3. Virtual Copy Constructors

Mammal11.cpp

```
#include <iostream>

class Mammal
{
	public:
		Mammal():age(1) { std::cout << "Mammal constructor ...\n"; }
		virtual ~Mammal() { std::cout << "Mammal destructor ...\n"; }
		Mammal (const Mammal &rhs);
		virtual void speak() const { std::cout << "Mammal speak!\n"; }
		virtual Mammal* clone() { return new Mammal(*this); }
		int getAge() const { return age; }

	protected:
		int age;
};

Mammal::Mammal (const Mammal &rhs):age(rhs.getAge())
{
	std::cout << "Mammal copy constructor ...\n";
}

```

…

```
class Dog : public Mammal
{
      public:
            Dog() { std::cout << “Dog constructor ...\n”; }
            virtual ~Dog() { std::cout << “Dog destructor ...\n”; }
            Dog (const Dog &rhs);
            void speak() const { std::cout << “Woof!\n”; }
            virtual Mammal* clone() { return new Dog(*this); }
};

Dog::Dog(const Dog &rhs):
Mammal(rhs)
{
      std::cout << “Dog copy constructor ...\n”;
}

class Cat : public Mammal
{
      public:
            Cat() { std::cout << “Cat constructor ...\n”; }
            virtual ~Cat() { std::cout << “Cat destructor ...\n”; }
            Cat (const Cat&);
            void speak() const { std::cout << “Meow!\n”; }
            virtual Mammal* Clone() { return new Cat(*this); }
};

Cat::Cat(const Cat &rhs):
Mammal(rhs)
{
      std::cout << “Cat copy constructor ...\n”;
}

enum ANIMALS { MAMMAL, DOG, CAT};
const int numAnimalTypes = 3;
```

…

```
int main()
{
        Mammal *array[numAnimalTypes];
        Mammal *ptr;
        int choice, i;
        for (i = 0; i < numAnimalTypes; i++)
        {
                std::cout << “(1) dog (2) cat (3) mammal: “;
                std::cin >> choice;
                switch (choice)
                {
                        case DOG:
                        ptr = new Dog;
                        break;
                        case CAT:
                        ptr = new Cat;
                        break;
                        default:
                        ptr = new Mammal;
                        break;
                }
                array[i] = ptr;
        }
        Mammal *otherArray[numAnimalTypes];
        for (i=0; i < numAnimalTypes; i++)
        {
                array[i]->speak();
                otherArray[i] = array[i]->clone();
        }
        for (i=0; i < numAnimalTypes; i++)
        {
                otherArray[i]->speak();
        }
        return 0;
}
```

4. Problems with Single Inheritance

Mammal12.cpp

```
#include <iostream>

class Mammal
{
        public:
                Mammal():age(1) { std::cout « 'Mammal constructor ...\n"; }
                virtual -Mammal() { std::cout « 'Mammal destructor ...\n'; }
                virtual void speak() const { std::cout « 'Mammal speakflnn; }
        protected:
        int age;
};

class Cat : public Mammal
{
        public:
                Cat() { std::cout << "Cat constructor ...\n"; }
                ~Cat() { std::cout << "Cat destructor ...\n"; }
                void speak() const { std::cout << "Meow!\n"; }
};

int main()
{
        Mammal *pCat = new Cat;
        pCat->speak();
        return 0;
}
```

Mammal13.cpp

```cpp
#include <iostream>

class Mammal
{
        public:
                Mammal():age(1) { std::cout << “Mammal constructor ...\n”; }
                virtual ~Mammal() { std::cout << “Mammal destructor ...\n”; }
                virtual void speak() const { std::cout << “Mammal speak!\n”; }
                protected:
                int age;
};

class Cat: public Mammal
{
        public:
                Cat() { std::cout << “Cat constructor ...\n”; }
                ~Cat() { std::cout << “Cat destructor ...\n”; }
                void speak() const { std::cout << “Meow!\n”; }
                void purr() const { std::cout << “Rrrrrrrrrrr!\n”; }
};

class Dog: public Mammal
{
        public:
                Dog() { std::cout << “Dog constructor ...\n”; }
                ~Dog() { std::cout << “Dog destructor ...\n”; }
                void speak() const { std::cout << “Woof!\n”; }
};

```

…

```
int main()
{
    const int numberMammals = 3;
    Mammal* zoo[numberMammals];
    Mammal* pMammal;
    int choice, i;
    for (i = 0; i < numberMammals; i++)
    {
        std::cout << “(1)Dog (2)Cat: “;
        std::cin >> choice;
        if (choice == 1)
            pMammal = new Dog;
        else
            pMammal = new Cat;

        zoo[i] = pMammal;
    }

    std::cout << “\n”;

    for (i = 0; i < numberMammals; i++)
    {
        zoo[i]->speak();

        Cat *pRealCat = dynamic_cast<Cat *> (zoo[i]);
        if (pRealCat)
            pRealCat->purr();
        else
            std::cout << “Uh oh, not a cat!\n”;

        delete zoo[i];
        std::cout << “\n”;
    }

    return 0;
}
```

5. Abstract Data Types

Shape.cpp

```cpp
#include <iostream>

class Shape
{
    public:
        Shape() {}
        virtual ~Shape() {}
        virtual long getArea() { return -1; } // error
        virtual long getPerim() { return -1; }
        virtual void draw() {}
};

class Circle : public Shape
{
    public:
        Circle(int newRadius):radius(newRadius) {}
        ~Circle() {}
        long getArea() { return 3 * radius * radius; }
        long getPerim() { return 9 * radius; }
        void draw();
    private:
        int radius;
        int circumference;
};

```

…

```
void Circle::draw()
{
        std::cout << "Circle drawing routine here!\n";
}

class Rectangle : public Shape
{
        public:
                Rectangle(int newLen, int newWidth):
                length(newLen), width(newWidth) {}
                virtual ~Rectangle() {}
                virtual long getArea() { return length * width; }
                virtual long getPerim() { return 2 * length + 2 * width; }
                virtual int getLength() { return length; }
                virtual int getWidth() { return width; }
                virtual void draw();
        private:
                int width;
                int length;
};

```

…

```
void Rectangle::draw()
{
        for (int i = 0; i < length; i++)
        {
                for (int j = 0; j < width; j++)
                        std::cout << “x “;

                std::cout << “\n”;
        }
}

class Square : public Rectangle
{
        public:
                Square(int len);
                Square(int len, int width);
                ~Square() {}
                long getPerim() { return 4 * getLength(); }
};

Square::Square(int newLen):
Rectangle(newLen, newLen)
{}

Square::Square(int newLen, int newWidth):
Rectangle(newLen, newWidth)
{
        if (getLength() != getWidth())
                std::cout << “Error, not a square ... a rectangle?\n”;
}

```

…

```cpp
int main()
{
        int choice;
        bool fQuit = false;
        Shape * sp;

        while (1)
        {
                std::cout << “(1) Circle (2) Rectangle (3) Square (0) Quit: ”;
                std::cin >> choice;

                switch (choice)
                {
                        case 1:
                        sp = new Circle(5);
                        break;
                        case 2:
                        sp = new Rectangle(4, 6);
                        break;
                        case 3:
                        sp = new Square(5);
                        break;
                        default:
                        fQuit = true;
                        break;
                }
                if (fQuit)
                        break;

                sp->draw();
                std::cout << “\n”;
        }
        return 0;
}
```

6. Implementing Pure Virtual Functions

Shape2.cpp

```
#include <iostream>

class Shape
{
        public:
                Shape() {}
                virtual ~Shape() {}
                virtual long getArea() = 0;
                virtual long getPerim()= 0;
                virtual void draw() = 0;
        private:
};

void Shape::draw()
{
        std::cout << “Abstract drawing mechanism!\n”;
}

class Circle : public Shape
{
        public:
                Circle(int newRadius):radius(newRadius) {}
                ~Circle() {}
                long getArea() { return 3 * radius * radius; }
                long getPerim() { return 9 * radius; }
                void draw();
        private:
                int radius;
                int circumference;
};

```

…

```
void Circle::draw()
{
        std::cout << "Circle drawing routine here!\n";
        Shape::draw();
}

class Rectangle : public Shape
{
        public:
                Rectangle(int newLen, int newWidth):
                length(newLen), width(newWidth) {}
                virtual ~Rectangle() {}
                long getArea() { return length * width; }
                long getPerim() { return 2 * length + 2 * width; }
                virtual int getLength() { return length; }
                virtual int getWidth() { return width; }
                void draw();
        private:
                int width;
                int length;
};

void Rectangle::draw()
{
        for (int i = 0; i < length; i++)
        {
                for (int j = 0; j < width; j++)
                        std::cout << "x ";

                std::cout << "\n";
        }
        Shape::draw();
}

```

…

```
class Square : public Rectangle
{
        public:
                Square(int len);
                Square(int len, int width);
                ~Square() {}
                long getPerim() {return 4 * getLength();}
};

Square::Square(int newLen):
Rectangle(newLen, newLen)
{}

Square::Square(int newLen, int newWidth):
Rectangle(newLen, newWidth)
{
        if (getLength() != getWidth())
                std::cout << “Error, not a square ... a rectangle?\n”;
}

```

…

```
int main()
{
    int choice;
    bool fQuit = false;
    Shape * sp;

    while (1)
    {
        std::cout << “(1) Circle (2) Rectangle (3) Square (0) Quit: “;
        std::cin >> choice;

        switch (choice)
        {
            case 1:
            sp = new Circle(5);
            break;
            case 2:
            sp = new Rectangle(4, 6);
            break;
            case 3:
            sp = new Square(5);
            break;
            default:
            fQuit = true;
            break;
        }
        if (fQuit)
            break;
        sp->draw();
        std::cout << “\n”;
    }
    return 0;
}
```

7. Complex Hierarchies of Abstraction

Animal.cpp

```
#include <iostream>

enum COLOR { Red, Green, Blue, Yellow, White, Black, Brown } ;

class Animal // common base to both horse and bird
{
	public:
		Animal(int);
		virtual ~Animal() { std::cout << "Animal destructor ...\n"; }
		virtual int getAge() const { return age; }
		virtual void setAge(int newAge) { age = newAge; }
		virtual void sleep() const = 0;
		virtual void eat() const = 0;
		virtual void reproduce() const = 0;
		virtual void move() const = 0;
		virtual void speak() const = 0;
	private:
		int age;
};

Animal::Animal(int newAge):
age(newAge)
{
	std::cout << "Animal constructor ...\n";
}

```

…

```
class Mammal : public Animal
{
      public:
            Mammal(int newAge):Animal(newAge)
            { std::cout << “Mammal constructor ...\n”;}
            virtual ~Mammal() { std::cout << “Mammal destructor ...\n”;}
            virtual void reproduce() const
            { std::cout << “Mammal reproduction depicted ...\n”; }
};

class Fish : public Animal
{
      public:
            Fish(int newAge):Animal(newAge)
            { std::cout << “Fish constructor ...\n”;}
            virtual ~Fish()
            { std::cout << “Fish destructor ...\n”; }
            virtual void sleep() const
            { std::cout << “Fish snoring ...\n”; }
            virtual void eat() const
            { std::cout << “Fish feeding ...\n”; }
            virtual void reproduce() const
            { std::cout << “Fish laying eggs ...\n”; }
            virtual void move() const
            { std::cout << “Fish swimming ...\n”; }
            virtual void speak() const { }
};

```

…

```
class Horse : public Mammal
{
        public:
                Horse(int newAge, COLOR newColor):
                Mammal(newAge), color(newColor)
                { std::cout << “Horse constructor ...\n”; }
                virtual ~Horse()
                { std::cout << “Horse destructor ...\n”; }
                virtual void speak() const
                { std::cout << “Whinny!\n”; }
                virtual COLOR getcolor() const
                { return color; }
                virtual void sleep() const
                { std::cout << “Horse snoring ...\n”; }
                virtual void eat() const
                { std::cout << “Horse feeding ...\n”; }
                virtual void move() const
                { std::cout << “Horse running ...\n”;}

        protected:
                COLOR color;
};

```

…

```
class Dog : public Mammal
{
        public:
                Dog(int newAge, COLOR newColor ):
                Mammal(newAge), color(newColor)
                { std::cout << "Dog constructor ...\n"; }
                virtual ~Dog()
                { std::cout << "Dog destructor ...\n"; }
                virtual void speak() const
                { std::cout << "Whoof!\n"; }
                virtual void sleep() const
                { std::cout << "Dog snoring ...\n"; }
                virtual void eat() const
                { std::cout << "Dog eating ...\n"; }
                virtual void move() const
                { std::cout << "Dog running...\n"; }
                virtual void reproduce() const
                { std::cout << "Dogs reproducing ...\n"; }

        protected:
                COLOR color;
};

```

...

```
int main()
{
        Animal *pAnimal = 0;
        int choice;
        bool fQuit = false;

        while (1)
        {
                std::cout << “(1) Dog (2) Horse (3) Fish (0) Quit: “;
                std::cin >> choice;

                switch (choice)
                {
                        case 1:
                        pAnimal = new Dog(5, Brown);
                        break;
                        case 2:
                        pAnimal = new Horse(4, Black);
                        break;
                        case 3:
                        pAnimal = new Fish(5);
                        break;
                        default:
                        fQuit = true;
                        break;
                }
                if (fQuit)
                        break;

                pAnimal->speak();
                pAnimal->eat();
                pAnimal->reproduce();
                pAnimal->move();
                pAnimal->sleep();
                delete pAnimal;
                std::cout << “\n”;
        }
return 0;
}
```

Tugas

- Modifikasi program Mammal8.cpp dengan membuka comentar pada baris ke 21 pada method speak() di dalam dog.
- Modifikasi program Mammal10.cpp untuk menghilangkan virtual pada baris ke 8 definisi dari speak() pada class utama. Mengapa fungsi override tidak pernah dipanggil?
- Modifikasi program Animal.cpp untuk merepresentasikan sebuah object dari tipe Animal atau Mammal. Apa yang sebetulnya dilakukan oleh compiler dan mengapa?
- Modifikasi program Mammal13.cpp untuk melihat apa yang terjadi jika Anda menghilangkan "if" pada baris ke 55-58 dan memanggil fungsi purr() di semua aspek. Yang manakah object yang bekerja secara baik dan yang mana yang gagal?