

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

```
1: #include <iostream>
2:
3: class Mammal
4: {
5:     public:
6:         Mammal():age(1) { std::cout << "Mammal constructor ...\n"; }
7:         ~Mammal() { std::cout << "Mammal destructor ...\n"; }
8:         void move() const { std::cout << "Mammal, move one step\n"; }
9:         virtual void speak() const { std::cout << "Mammal speak!\n"; }
10:
11:     protected:
12:         int age;
13: };
14:
15: class Dog : public Mammal
16: {
17:     public:
18:         Dog() { std::cout << "Dog constructor ...\n"; }
19:         ~Dog() { std::cout << "Dog destructor ..\n"; }
20:         void wagTail() { std::cout << "Wagging tail ...\n"; }
21:         void speak() const { std::cout << "Woof!\n"; }
22:         void move() const { std::cout << "Dog moves 5 steps ...\n"; }
23: };
24:
25: int main()
26: {
27:     Mammal *pDog = new Dog;
28:     pDog->move();
29:     pDog->speak();
30:     return 0;
31: }
```

Mammal9.cpp

```
1: #include <iostream>
2:
3: class Mammal
4: {
5:     public:
6:         Mammal():age(1) { }
7:         ~Mammal() { }
8:         virtual void speak() const { std::cout << "Mammal speak!\n"; }
9:     protected:
10:        int age;
11: };
12:
13: class Dog : public Mammal
14: {
15:     public:
16:         void speak() const { std::cout << "Woof!\n"; }
17: };
18:
19: class Cat : public Mammal
20: {
21:     public:
22:         void speak() const { std::cout << "Meow!\n"; }
23: };
24:
25: class Horse : public Mammal
26: {
27:     public:
28:         void speak() const { std::cout << "Whinny!\n"; }
29: };
30:
31: class Pig : public Mammal
32: {
33:     public:
34:         void speak() const { std::cout << "Oink!\n"; }
35: };
36:
```

...

```
37: int main()
38: {
39:     Mammal* array[5];
40:     Mammal* ptr;
41:     int choice, i;
42:     for (i = 0; i < 5; i++)
43:     {
44:         std::cout << "(1) dog (2) cat (3) horse (4) pig: ";
45:         std::cin >> choice;
46:         switch (choice)
47:         {
48:             case 1:
49:                 ptr = new Dog;
50:                 break;
51:             case 2:
52:                 ptr = new Cat;
53:                 break;
54:             case 3:
55:                 ptr = new Horse;
56:                 break;
57:             case 4:
58:                 ptr = new Pig;
59:                 break;
60:             default:
61:                 ptr = new Mammal;
62:                 break;
63:         }
64:         array[i] = ptr;
65:     }
66:     for (i=0; i < 5; i++)
67:     {
68:         array[i]->speak();
69:     }
70:     return 0;
71: }
```

2. Slicing

Mammal10.cpp

```
1: #include <iostream>
2:
3: class Mammal
4: {
5:     public:
6:         Mammal():age(1) { }
7:         ~Mammal() { }
8:         virtual void speak() const { std::cout << "Mammal speak!\n"; }
9:     protected:
10:         int age;
11: };
12:
13: class Dog : public Mammal
14: {
15:     public:
16:         void speak() const { std::cout << "Woof!\n"; }
17: };
18:
19: class Cat : public Mammal
20: {
21:     public:
22:         void speak()const { std::cout << "Meow!\n"; }
23: };
24:
25: void valueFunction(Mammal);
26: void ptrFunction(Mammal*);
27: void refFunction(Mammal&);
28:
```


...

```
29: int main()
30: {
31:     Mammal* ptr=0;
32:     int choice;
33:     while (1)
34:     {
35:         bool fQuit = false;
36:         std::cout << "(1) dog (2) cat (0) quit: ";
37:         std::cin >> choice;
38:         switch (choice)
39:         {
40:             case 0:
41:                 fQuit = true;
42:                 break;
43:             case 1:
44:                 ptr = new Dog;
45:                 break;
46:             case 2:
47:                 ptr = new Cat;
48:                 break;
49:             default:
50:                 ptr = new Mammal;
51:                 break;
52:         }
53:         if (fQuit)
54:         {
55:             break;
56:         }
57:         ptrFunction(ptr);
58:         refFunction(*ptr);
59:         valueFunction(*ptr);
60:     }
61:     return 0;
62: }
63:
```

...

```
64: void valueFunction(Mammal mammalValue) // This function is called last
65: {
66:     mammalValue.speak();
67: }
68:
69: void ptrFunction (Mammal *pMammal)
70: {
71:     pMammal->speak();
72: }
73:
74: void refFunction (Mammal &rMammal)
75: {
76:     rMammal.speak();
77: }
```

3. Virtual Copy Constructors

Mammal11.cpp

```
1: #include <iostream>
2:
3: class Mammal
4: {
5:     public:
6:         Mammal():age(1) { std::cout << "Mammal constructor ...\n"; }
7:         virtual ~Mammal() { std::cout << "Mammal destructor ...\n"; }
8:         Mammal (const Mammal &rhs);
9:         virtual void speak() const { std::cout << "Mammal speak!\n"; }
10:        virtual Mammal* clone() { return new Mammal(*this); }
11:        int getAge() const { return age; }
12:
13:     protected:
14:         int age;
15: };
16:
17: Mammal::Mammal (const Mammal &rhs):age(rhs.getAge())
18: {
19:     std::cout << "Mammal copy constructor ...\n";
20: }
21:
```

...

```
22: class Dog : public Mammal
23: {
24:     public:
25:         Dog() { std::cout << "Dog constructor ...\n"; }
26:         virtual ~Dog() { std::cout << "Dog destructor ...\n"; }
27:         Dog (const Dog &rhs);
28:         void speak() const { std::cout << "Woof!\n"; }
29:         virtual Mammal* clone() { return new Dog(*this); }
30: };
31:
32: Dog::Dog(const Dog &rhs):
33: Mammal(rhs)
34: {
35:     std::cout << "Dog copy constructor ...\n";
36: }
37:
38: class Cat : public Mammal
39: {
40:     public:
41:         Cat() { std::cout << "Cat constructor ...\n"; }
42:         virtual ~Cat() { std::cout << "Cat destructor ...\n"; }
43:         Cat (const Cat&);
44:         void speak() const { std::cout << "Meow!\n"; }
45:         virtual Mammal* Clone() { return new Cat(*this); }
46: };
47:
48: Cat::Cat(const Cat &rhs):
49: Mammal(rhs)
50: {
51:     std::cout << "Cat copy constructor ...\n";
52: }
53:
54: enum ANIMALS { MAMMAL, DOG, CAT};
55: const int numAnimalTypes = 3;
```

...

```
56: int main()
57: {
58:     Mammal *array[numAnimalTypes];
59:     Mammal *ptr;
60:     int choice, i;
61:     for (i = 0; i < numAnimalTypes; i++)
62:     {
63:         std::cout << "(1) dog (2) cat (3) mammal: ";
64:         std::cin >> choice;
65:         switch (choice)
66:         {
67:             case DOG:
68:                 ptr = new Dog;
69:                 break;
70:             case CAT:
71:                 ptr = new Cat;
72:                 break;
73:             default:
74:                 ptr = new Mammal;
75:                 break;
76:         }
77:         array[i] = ptr;
78:     }
79:     Mammal *otherArray[numAnimalTypes];
80:     for (i=0; i < numAnimalTypes; i++)
81:     {
82:         array[i]->speak();
83:         otherArray[i] = array[i]->clone();
84:     }
85:     for (i=0; i < numAnimalTypes; i++)
86:     {
87:         otherArray[i]->speak();
88:     }
89:     return 0;
90: }
```

4. Problems with Single Inheritance

Mammal12.cpp

```
1: #include <iostream>
2:
3: class Mammal
4: {
5:     public:
6:         Mammal():age(1) { std::cout << 'Mammal constructor ...\n"; }
7:         virtual ~Mammal() { std::cout << 'Mammal destructor ...\n'; }
8:         virtual void speak() const { std::cout << 'Mammal speakflnn; }
9:     protected:
10:    int age;
11: };
12:
13: class Cat : public Mammal
14: {
15:     public:
16:         Cat() { std::cout << "Cat constructor ...\n"; }
17:         ~Cat() { std::cout << "Cat destructor ...\n"; }
18:         void speak() const { std::cout << "Meow!\n"; }
19: };
20:
21: int main()
22: {
23:     Mammal *pCat = new Cat;
24:     pCat->speak();
25:     return 0;
26: }
```

Mammal13.cpp

```
1: #include <iostream>
2:
3: class Mammal
4: {
5:     public:
6:         Mammal():age(1) { std::cout << "Mammal constructor ...\n"; }
7:         virtual ~Mammal() { std::cout << "Mammal destructor ...\n"; }
8:         virtual void speak() const { std::cout << "Mammal speak!\n"; }
9:     protected:
10:        int age;
11: };
12:
13: class Cat: public Mammal
14: {
15:     public:
16:         Cat() { std::cout << "Cat constructor ...\n"; }
17:         ~Cat() { std::cout << "Cat destructor ...\n"; }
18:         void speak() const { std::cout << "Meow!\n"; }
19:         void purr() const { std::cout << "Rrrrrrrrrrr!\n"; }
20: };
21:
22: class Dog: public Mammal
23: {
24:     public:
25:         Dog() { std::cout << "Dog constructor ...\n"; }
26:         ~Dog() { std::cout << "Dog destructor ...\n"; }
27:         void speak() const { std::cout << "Woof!\n"; }
28: };
29:
```

...

```
30: int main()
31: {
32:     const int numberMammals = 3;
33:     Mammal* zoo[numberMammals];
34:     Mammal* pMammal;
35:     int choice, i;
36:     for (i = 0; i < numberMammals; i++)
37:     {
38:         std::cout << "(1)Dog (2)Cat: ";
39:         std::cin >> choice;
40:         if (choice == 1)
41:             pMammal = new Dog;
42:         else
43:             pMammal = new Cat;
44:
45:         zoo[i] = pMammal;
46:     }
47:
48:     std::cout << "\n";
49:
50:     for (i = 0; i < numberMammals; i++)
51:     {
52:         zoo[i]->speak();
53:
54:         Cat *pRealCat = dynamic_cast<Cat *> (zoo[i]);
55:         if (pRealCat)
56:             pRealCat->purr();
57:         else
58:             std::cout << "Uh oh, not a cat!\n";
59:
60:         delete zoo[i];
61:         std::cout << "\n";
62:     }
63:
64:     return 0;
65: }
```


5. Abstract Data Types

Shape.cpp

```
1: #include <iostream>
2:
3: class Shape
4: {
5:     public:
6:         Shape() {}
7:         virtual ~Shape() {}
8:         virtual long getArea() { return -1; } // error
9:         virtual long getPerim() { return -1; }
10:        virtual void draw() {}
11: };
12:
13: class Circle : public Shape
14: {
15:     public:
16:         Circle(int newRadius):radius(newRadius) {}
17:         ~Circle() {}
18:         long getArea() { return 3 * radius * radius; }
19:         long getPerim() { return 9 * radius; }
20:         void draw();
21:     private:
22:         int radius;
23:         int circumference;
24: };
25:
```

...

```
26: void Circle::draw()
27: {
28:     std::cout << "Circle drawing routine here!\n";
29: }
30:
31: class Rectangle : public Shape
32: {
33:     public:
34:         Rectangle(int newLen, int newWidth):
35:             length(newLen), width(newWidth) {}
36:         virtual ~Rectangle() {}
37:         virtual long getArea() { return length * width; }
38:         virtual long getPerim() { return 2 * length + 2 * width; }
39:         virtual int getLength() { return length; }
40:         virtual int getWidth() { return width; }
41:         virtual void draw();
42:     private:
43:         int width;
44:         int length;
45: };
46:
```

...

```
47: void Rectangle::draw()
48: {
49:     for (int i = 0; i < length; i++)
50:     {
51:         for (int j = 0; j < width; j++)
52:             std::cout << "x ";
53:
54:         std::cout << "\n";
55:     }
56: }
57:
58: class Square : public Rectangle
59: {
60:     public:
61:         Square(int len);
62:         Square(int len, int width);
63:         ~Square() {}
64:         long getPerim() { return 4 * getLength(); }
65: };
66:
67: Square::Square(int newLen):
68: Rectangle(newLen, newLen)
69: {}
70:
71: Square::Square(int newLen, int newWidth):
72: Rectangle(newLen, newWidth)
73: {
74:     if (getLength() != getWidth())
75:         std::cout << "Error, not a square ... a rectangle?\n";
76: }
77:
```

...

```
78: int main()
79: {
80:     int choice;
81:     bool fQuit = false;
82:     Shape * sp;
83:
84:     while (1)
85:     {
86:         std::cout << "(1) Circle (2) Rectangle (3) Square (0) Quit: ";
87:         std::cin >> choice;
88:
89:         switch (choice)
90:         {
91:             case 1:
92:                 sp = new Circle(5);
93:                 break;
94:             case 2:
95:                 sp = new Rectangle(4, 6);
96:                 break;
97:             case 3:
98:                 sp = new Square(5);
99:                 break;
100:            default:
101:                fQuit = true;
102:                break;
103:        }
104:        if (fQuit)
105:            break;
106:
107:        sp->draw();
108:        std::cout << "\n";
109:    }
110:    return 0;
111: }
```

6. Implementing Pure Virtual Functions

Shape2.cpp

```
1: #include <iostream>
2:
3: class Shape
4: {
5:     public:
6:         Shape() {}
7:         virtual ~Shape() {}
8:         virtual long getArea() = 0;
9:         virtual long getPerim()= 0;
10:        virtual void draw() = 0;
11:    private:
12: };
13:
14: void Shape::draw()
15: {
16:     std::cout << "Abstract drawing mechanism!\n";
17: }
18:
19: class Circle : public Shape
20: {
21:     public:
22:         Circle(int newRadius):radius(newRadius) {}
23:         ~Circle() {}
24:         long getArea() { return 3 * radius * radius; }
25:         long getPerim() { return 9 * radius; }
26:         void draw();
27:     private:
28:         int radius;
29:         int circumference;
30: };
31:
```

...

```
32: void Circle::draw()
33: {
34:     std::cout << "Circle drawing routine here!\n";
35:     Shape::draw();
36: }
37:
38: class Rectangle : public Shape
39: {
40:     public:
41:         Rectangle(int newLen, int newWidth):
42:             length(newLen), width(newWidth) {}
43:         virtual ~Rectangle() {}
44:         long getArea() { return length * width; }
45:         long getPerim() { return 2 * length + 2 * width; }
46:         virtual int getLength() { return length; }
47:         virtual int getWidth() { return width; }
48:         void draw();
49:     private:
50:         int width;
51:         int length;
52: };
53:
54: void Rectangle::draw()
55: {
56:     for (int i = 0; i < length; i++)
57:     {
58:         for (int j = 0; j < width; j++)
59:             std::cout << "x ";
60:
61:         std::cout << "\n";
62:     }
63:     Shape::draw();
64: }
65:
```

...

```
66: class Square : public Rectangle
67: {
68:     public:
69:         Square(int len);
70:         Square(int len, int width);
71:         ~Square() {}
72:         long getPerim() {return 4 * getLength();}
73: };
74:
75: Square::Square(int newLen):
76: Rectangle(newLen, newLen)
77: {}
78:
79: Square::Square(int newLen, int newWidth):
80: Rectangle(newLen, newWidth)
81: {
82:     if (getLength() != getWidth())
83:         std::cout << "Error, not a square ... a rectangle?\n";
84: }
85:
```

...

```
86: int main()
87: {
88:     int choice;
89:     bool fQuit = false;
90:     Shape * sp;
91:
92:     while (1)
93:     {
94:         std::cout << "(1) Circle (2) Rectangle (3) Square (0) Quit: ";
95:         std::cin >> choice;
96:
97:         switch (choice)
98:         {
99:             case 1:
100:                sp = new Circle(5);
101:                break;
102:             case 2:
103:                sp = new Rectangle(4, 6);
104:                break;
105:             case 3:
106:                sp = new Square(5);
107:                break;
108:             default:
109:                fQuit = true;
110:                break;
111:         }
112:         if (fQuit)
113:             break;
114:         sp->draw();
115:         std::cout << "\n";
116:     }
117:     return 0;
118: }
```


7. Complex Hierarchies of Abstraction

Animal.cpp

```
1: #include <iostream>
2:
3: enum COLOR { Red, Green, Blue, Yellow, White, Black, Brown } ;
4:
5: class Animal // common base to both horse and bird
6: {
7:     public:
8:         Animal(int);
9:         virtual ~Animal() { std::cout << "Animal destructor ...\n"; }
10:        virtual int getAge() const { return age; }
11:        virtual void setAge(int newAge) { age = newAge; }
12:        virtual void sleep() const = 0;
13:        virtual void eat() const = 0;
14:        virtual void reproduce() const = 0;
15:        virtual void move() const = 0;
16:        virtual void speak() const = 0;
17:    private:
18:        int age;
19: };
20:
21: Animal::Animal(int newAge):
22: age(newAge)
23: {
24:     std::cout << "Animal constructor ...\n";
25: }
26:
```

...

```
27: class Mammal : public Animal
28: {
29:     public:
30:         Mammal(int newAge):Animal(newAge)
31:         { std::cout << "Mammal constructor ...\n";}
32:         virtual ~Mammal() { std::cout << "Mammal destructor ...\n";}
33:         virtual void reproduce() const
34:         { std::cout << "Mammal reproduction depicted ...\n"; }
35: };
36:
37: class Fish : public Animal
38: {
39:     public:
40:         Fish(int newAge):Animal(newAge)
41:         { std::cout << "Fish constructor ...\n";}
42:         virtual ~Fish()
43:         { std::cout << "Fish destructor ...\n"; }
44:         virtual void sleep() const
45:         { std::cout << "Fish snoring ...\n"; }
46:         virtual void eat() const
47:         { std::cout << "Fish feeding ...\n"; }
48:         virtual void reproduce() const
49:         { std::cout << "Fish laying eggs ...\n"; }
50:         virtual void move() const
51:         { std::cout << "Fish swimming ...\n"; }
52:         virtual void speak() const { }
53: };
54:
```

...

```
55: class Horse : public Mammal
56: {
57:     public:
58:         Horse(int newAge, COLOR newColor):
59:             Mammal(newAge), color(newColor)
60:         { std::cout << "Horse constructor ...\n"; }
61:         virtual ~Horse()
62:         { std::cout << "Horse destructor ...\n"; }
63:         virtual void speak() const
64:         { std::cout << "Whinny!\n"; }
65:         virtual COLOR getcolor() const
66:         { return color; }
67:         virtual void sleep() const
68:         { std::cout << "Horse snoring ...\n"; }
69:         virtual void eat() const
70:         { std::cout << "Horse feeding ...\n"; }
71:         virtual void move() const
72:         { std::cout << "Horse running ...\n"; }
73:
74:     protected:
75:         COLOR color;
76: };
77:
```

...

```
78: class Dog : public Mammal
79: {
80:     public:
81:         Dog(int newAge, COLOR newColor ):
82:             Mammal(newAge), color(newColor)
83:         { std::cout << "Dog constructor ...\n"; }
84:         virtual ~Dog()
85:         { std::cout << "Dog destructor ...\n"; }
86:         virtual void speak() const
87:         { std::cout << "Whoof!\n"; }
88:         virtual void sleep() const
89:         { std::cout << "Dog snoring ...\n"; }
90:         virtual void eat() const
91:         { std::cout << "Dog eating ...\n"; }
92:         virtual void move() const
93:         { std::cout << "Dog running...\n"; }
94:         virtual void reproduce() const
95:         { std::cout << "Dogs reproducing ...\n"; }
96:
97:     protected:
98:         COLOR color;
99: };
100:
```

...

```
101: int main()
102: {
103:     Animal *pAnimal = 0;
104:     int choice;
105:     bool fQuit = false;
106:
107:     while (1)
108:     {
109:         std::cout << "(1) Dog (2) Horse (3) Fish (0) Quit: ";
110:         std::cin >> choice;
111:
112:         switch (choice)
113:         {
114:             case 1:
115:                 pAnimal = new Dog(5, Brown);
116:                 break;
117:             case 2:
118:                 pAnimal = new Horse(4, Black);
119:                 break;
120:             case 3:
121:                 pAnimal = new Fish(5);
122:                 break;
123:             default:
124:                 fQuit = true;
125:                 break;
126:         }
127:         if (fQuit)
128:             break;
129:
130:         pAnimal->Speak();
131:         pAnimal->eat();
132:         pAnimal->reproduce();
133:         pAnimal->move();
134:         pAnimal->sleep();
135:         delete pAnimal;
136:         std::cout << "\n";
137:     }
138:     return 0;
139: }
```

Tugas

- Modifikasi program Mammal8.cpp dengan membuka komentar 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?