KIE1008: Programming 2

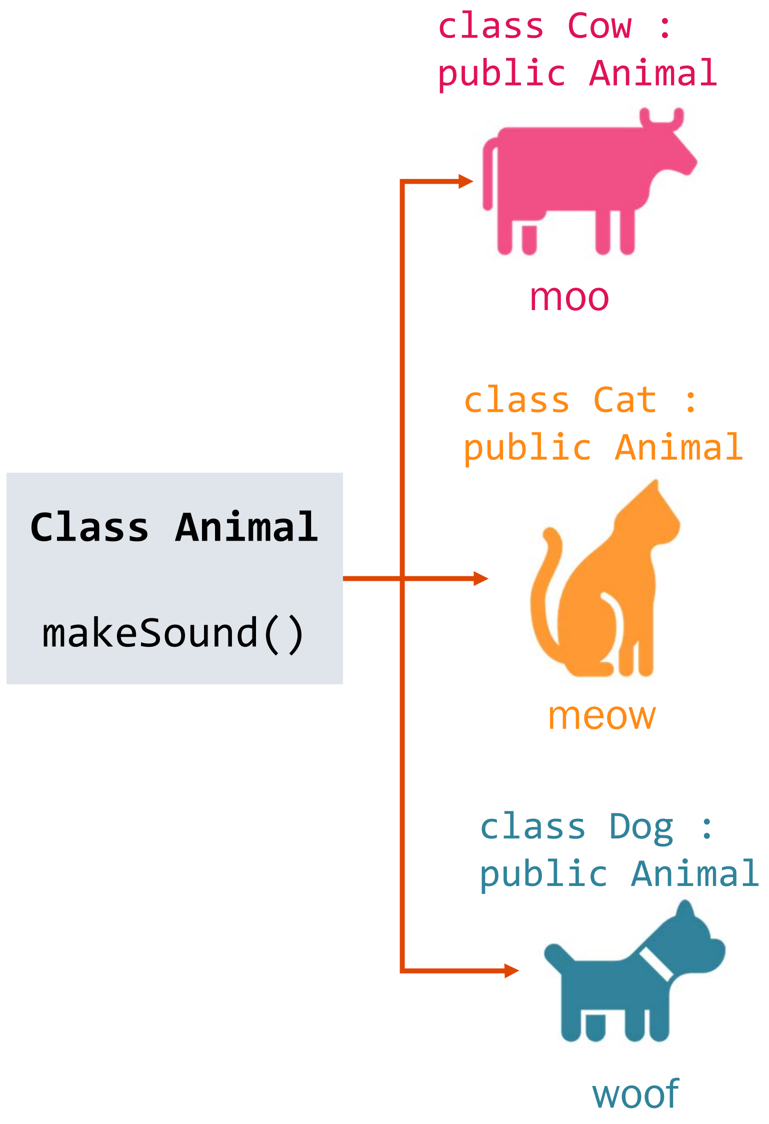
Object-Oriented Programming: Polymorphism

Semester 1, 2025/2026

Aydon.

Polymorphism

- Polymorphism enables programmers to "program in the general" rather than "program in the specific".
- Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.
- The key concept of polymorphism is relying on each object to know how to "do the right thing" in response to the same function call.



Polymorphism

- With polymorphism, programmers can design and implement systems that are easily extensible.
  - New classes can be added with little or no modification to the general portions of the program, as long as the new classes are part of the inheritance hierarchy that the program processes generally.
  - The only parts of a program that must be altered to accommodate new classes are those that require direct knowledge of the new classes that you add to the hierarchy.
- Polymorphism works on object pointers and references using so-called dynamic binding at run-time. It does not work on regular objects, which uses static binding during the compile-time.

Example: Need of Polymorphism (1/4)

```
class baseClass
{
public:
  baseClass(int val) :x(val) {  }
  void info()
  {
    cout << "Info member function of base class" << endl;
  }
protected:
  int x;
};
```

Code: W06C01

Example: Need of Polymorphism (2/4)

```cpp
class derivedClass1 : public baseClass
{
public:
  derivedClass1(int val) :baseClass(val) {  }
  void info() {
    cout << "Info member function of derived class 1" << endl;
  }
};


class derivedClass2 : public baseClass
{
public:
  derivedClass2(int val) :baseClass(val) { }
  void info() {
    cout << "Info member function of derived class 2" << endl;
  }
};
```

Code: W06C01

Example: Need of Polymorphism (3/4)

```
int main()
{
   baseClass b(1); //create base-class object

   derivedClass1 d1(2); //create derived-class object
   derivedClass2 d2(3); //create derived-class object

   baseClass *bPtr1 = nullptr; //creates a base-class pointer
   baseClass *bPtr2 = nullptr; //creates a base-class pointer
   derivedClass1 *d1Ptr = nullptr; //creates a derived-class pointer
   derivedClass2 *d2Ptr = nullptr; //creates a derived-class pointer

   bPtr1 = &b; //Aiming a base-class pointer at a base-class object
   bPtr1 -> info(); //base-class member function is invoked
```

Code: W06C01

Example: Need of Polymorphism (4/4)

```
    d1Ptr  =  &d1;  //Aiming  a  derived-class  pointer  at  a  derived-class
                       object
    d1Ptr -> info(); //derived-class member function is invoked

    //Aiming base-class pointers to derived class objects (upcast)
    bPtr1 = &d1;
    bPtr2 = &d2;

    //Calling info function
    bPtr1->info(); //base-class member function is invoked
    bPtr2->info(); //base-class member function is invoked
}
```

The base class function is called irrespective of the type of object pointed at by the pointer. Solution?

Code: W06C01

Virtual Functions

Virtual functions are functions which are expected to be overridden in the derived class. By using `virtual` function, we can call functions of a derived class using pointer of the base class.

<u>Example</u>

Modify function `info()` of the `baseClass` in Example 1 as virtual function and run the codes again.

```
class baseClass {
......
virtual void info()  {
    cout << "Info member function of base class" << endl;
  }
};
```

Virtual Functions

- The keyword `virtual` determines which method is used if the method is invoked by a pointer (or reference).
- Without `virtual`, the program chooses the method based on the pointer type; with `virtual`, the program chooses the method based on the type of the object pointed-to.
- If you override function in the derived class, the overridden function shall have the same parameter list as the base class' version.

Invoking a `virtual` Function

(A) Through a Base-Class Pointer or Reference

- e.g., `shapePtr->draw()`, `shapeRef.draw()`
- dynamic binding (or late binding): the program will choose the correct derived-class draw function dynamically (i.e., at execution time) based on the object type – not the pointer or reference type.

(B) Through an Object’s Name

- e.g., `squareObject.draw()`
- static binding: the function invocation is resolved at compile time – this is not polymorphic behavior.

`virtual` **Destructors**

- If a class has `virtual` functions, always provide a `virtual` destructor, even if one is not required for the class. This ensures that a custom derived-class destructor (if there is one) will be invoked when a derived-class object is deleted via a base class pointer.
- If a base class destructor is declared `virtual`, the destructors of any derived classes are also `virtual` and they override the base class destructor.
- Constructor can't be `virtual`, because it is not inherited.

Example: `virtual` Destructors

```cpp
class baseClass {
public:
    baseClass() { cout << "Constructing base \n"; }
    virtual ~baseClass() { cout << "Destructing base \n"; }
};

class derivedClass: public baseClass {
public:
    derivedClass() { cout << "Constructing derived \n"; }
    ~derivedClass() { cout << "Destructing derived \n"; }
};

int main() {
  derivedClass *d = new derivedClass();
  baseClass *b = d;
  delete b;
  return 0;
}
```

Code: W06C02

Abstract Classes

- An abstract class is a class from which you never intend to instantiate any objects. Because these classes normally are used as base classes in inheritance hierarchies, we refer to them as abstract base classes.
- Abstract classes are incomplete—derived classes must define the “missing pieces” before objects of these classes can be instantiated.

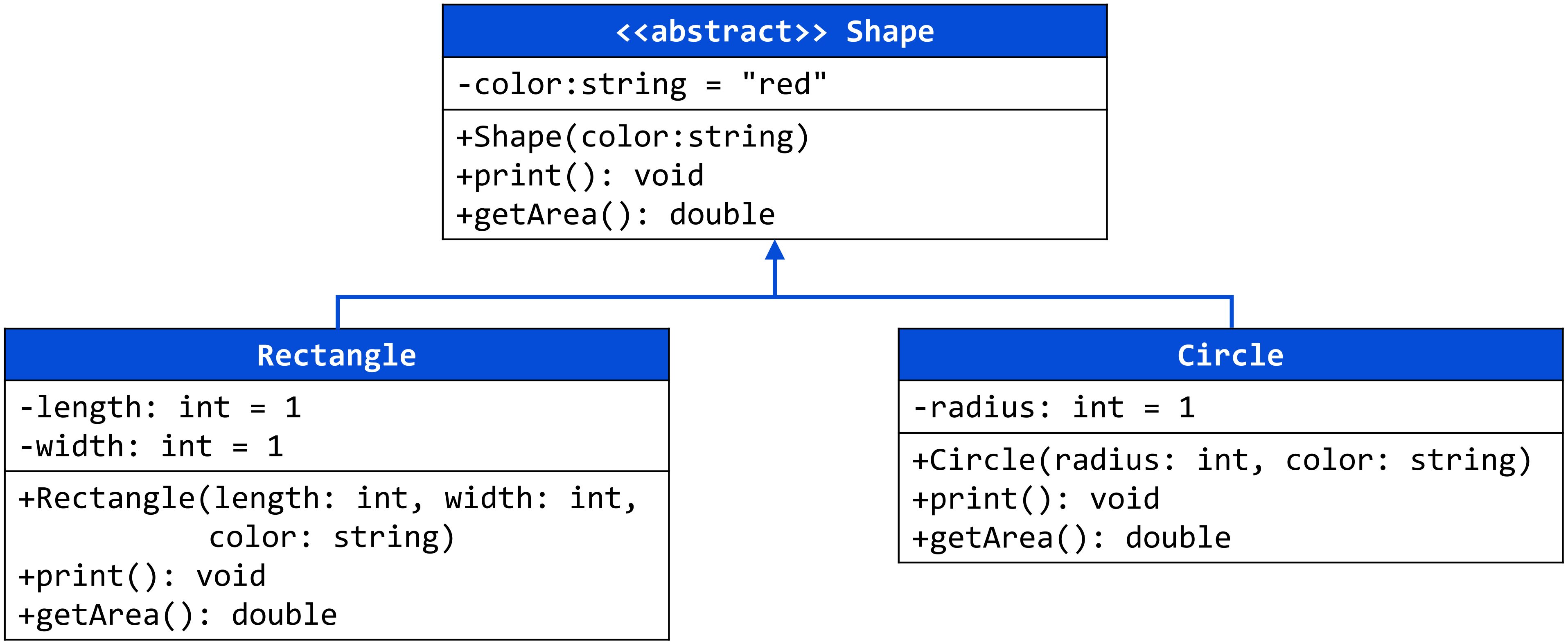
Pure `virtual` Functions

- A class is made abstract by declaring one or more of its virtual functions to be “pure”.
- A pure virtual function is specified by placing “= 0” (pure specifier) in its declaration:

```
virtual void func1() const = 0;
```

- Pure virtual functions typically do not provide implementations, though they can.

Example:
Abstract Classes and Pure `virtual` Functions



Example: Abstract Classes and Pure `virtual` Functions (1/6)

```cpp
class Shape { //abstract base class
private:
   string color;
public:
   Shape(string color = "red");  // Constructor
   virtual void print() const; // Virtual function
   virtual double getArea() const = 0; // Pure virtual function
};

Shape::Shape(string color) {
   this->color = color;
}

void Shape::print() const {
   cout << "Shape of color = " << color;
}
```

Code: W06C03

Example: Abstract Classes and Pure `virtual` Functions (2/6)

```cpp
class Circle : public Shape {
private:
   int radius;
public:
   Circle(int r = 1, string color = "red");  // Constructor
   void print() const; // Override virtual function
   double getArea() const; // Implement virtual function
};

Circle::Circle(int r, string color)
: Shape(color), radius(r) { }

void Circle::print() const {
   cout << "Circle radius = " << radius << ", derived class of ";
   Shape::print();
}
```

Code: W06C03

Example: Abstract Classes and Pure `virtual` Functions (3/6)

```cpp
double Circle::getArea() const {
    return radius * radius * 3.14159265;
}

class Rectangle : public Shape {
private:
   int length, width;
public:
   Rectangle(int l = 1, int w = 1, string color = "red");
   void print() const; // Override virtual function
   double getArea() const;  // Implement virtual function
};
```

Code: W06C03

Example: Abstract Classes and Pure `virtual` Functions (4/6)

```cpp
Rectangle::Rectangle(int l, int w, string color)
: Shape(color), length(l), width(w) { }

void Rectangle::print() const {
    cout << "Rectangle length=" << length << " width=" << width <<
    ", derived class of ";
    Shape::print();
}

double Rectangle::getArea() const {
    return length * width;
}
```

Code: W06C03

Example: Abstract Classes and Pure `virtual` Functions (5/6)

```cpp
int main() {

// Shape s1;  //Cannot create instance of abstract class Shape
   Shape * s1, * s2;  //Shape pointers

   s1 = new Circle(6);
   s1->print();
   cout << endl;
   cout << "Area of s1 = " << s1->getArea() << endl;

   s2 = new Rectangle(7, 8);
   s2->print();
   cout << endl;
   cout << "Area of s2 = " << s2->getArea() << endl;

   delete s1;
   delete s2;
```

Code: W06C03

Example: Abstract Classes and Pure `virtual` Functions (6/6)

```
// Shape s3 = Circle(6);  //Cannot allocate an object of abstract type
   Circle c3(8);
   Shape & s3 = c3; //Object reference
   s3.print();
   cout << endl;
   cout << "Area of s3 = " << s3.getArea() << endl;

   Circle c4(9);
   Shape * s4 = &c4; //Object pointer
   s4->print();
   cout << endl;
   cout << "Area of s4 = " << s4->getArea() << endl;
   return 0;
}
```

Code: W06C03

Example:
Abstract Classes and Pure `virtual` Functions

`Class`	`print()`	`getArea()`
`Shape`	`Shape of color =` *color*	`= 0`
`Circle`	`Circle radius =` *length*`, derived class of Shape of color =` *color*	`radius * radius * 3.14159265`
`Rectangle`	`Rectangle length =` *length* `width =` *width*`, derived class of Shape of color =` *color*	`length * width`

Case Study:
Payroll System Using Polymorphism

```cpp
int main()
{
    Employee *employees[3];

    employees[0] = &salariedEmployee;
    employees[1] = &commissionEmployee;
    employees[2] = &basePlusCommissionEmployee;

    for (const Employee *employeePtr : employees)
    {
        employeePtr->print();
        *employeePtr.print();
    }
}
```

Code: W06C04