Polymorphism

Prof. Jinkyu Jeong (Jinkyu@skku.edu)
TA -- Minwoo Ahn (minwoo.ahn@csl.skku.edu)
TA -- Donghyun Kim (donghyun.kim@csl.skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

Object-oriented Concept

- Encapsulation
  - ADT(Abstract data types), Object
- Inheritance
  - Derived object
- Polymorphism
  - Each object knows what it is

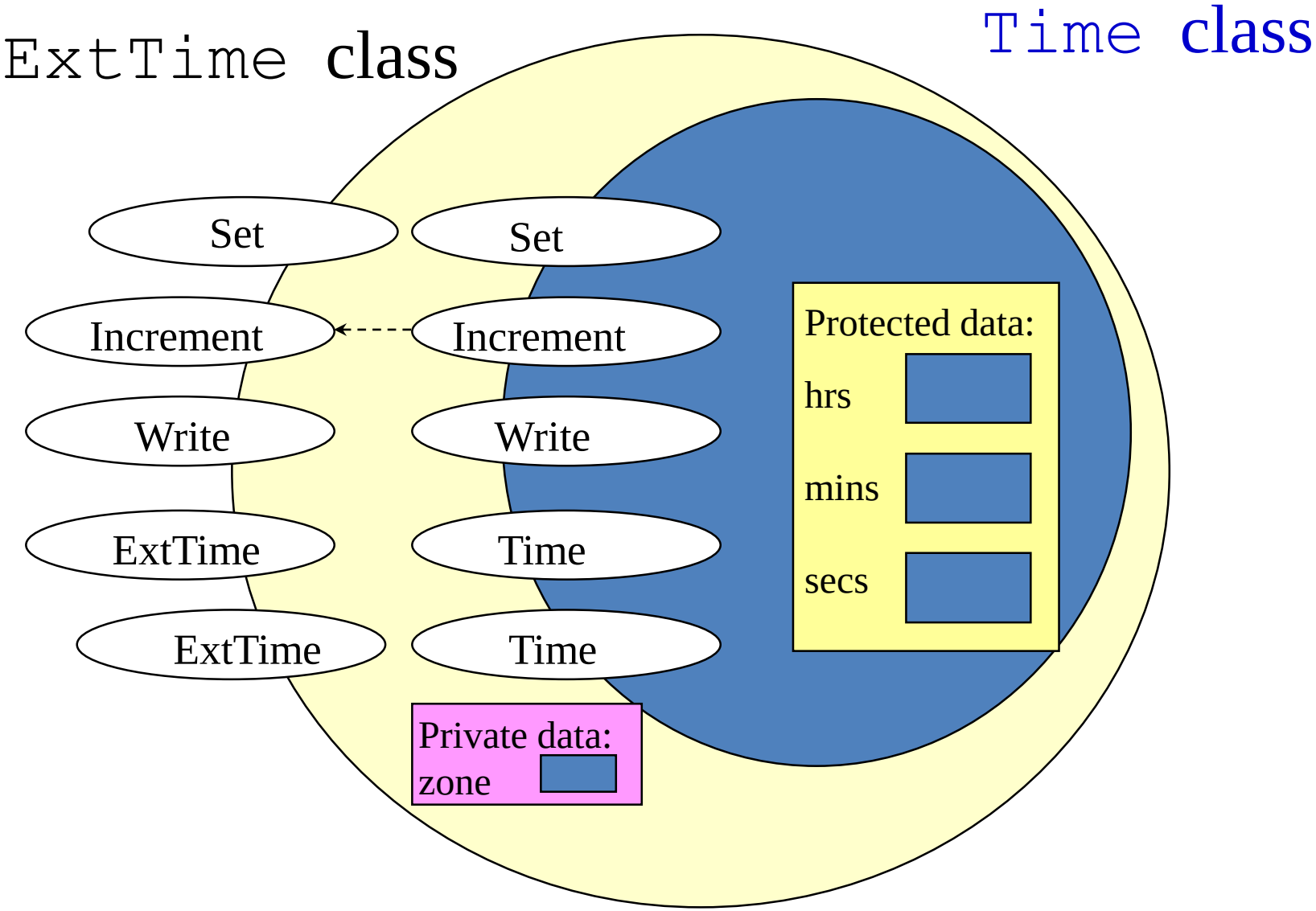
Polymorphism

- *noun, the quality or state of being able to assume different forms* – Webster
- An essential feature of an OO Language
- It builds upon "Inheritance"

Before we proceed…

- Inheritance – Basic Concepts
  - Class hierarchy
    - Code reuse, easy to maintain
  - Type of inheritance: public, private
  - Function overriding

Class Interface Diagram



Polymorphism: Example

```
class Dog {
    public:
     virtual void bark() = 0;
};
class 불독 : public Dog {
    public:
     void bark() {std::cout<<"왈왈"<<endl;}
};
class 진돗개 : public Dog {
    public:
     void bark() {std::cout<<"멍멍"<<endl;}
};
class 치와와 : public Dog {
    public:
     void bark() {std::cout<<"깽깽"<<endl;}
};
```

CLIENT CODE

```
void do_bark(Dog *dog) {
    dog->bark();
}

int main() {
    불독 Bulldog;
    진돗개 Jindog;
    치와와 Chiwawa;

    do_bark(&Bulldog);
    do_bark(&Jindog);
    do_bark(&Chiwawa);
}
```

OUTPUT

```
왈왈
멍멍
깽깽
```

Why Polymorphism?

```
void  Print  (Time   someTime )  //pass an object by value
{
     cout  <<  "Time is  " ;
     someTime.Write (  ) ;
     cout  <<  endl ;
}
```

```
// Time :: write()
```

CLIENT CODE

```
Time        startTime ( 8, 30, 0 ) ;
ExtTime     endTime (10, 45, 0, CST) ;

Print ( startTime ) ;
Print ( endTime ) ;
```

OUTPUT

```
Time is  08:30:00
Time is  19:45:00
```

Static Binding

- When the type of a formal parameter is a parent class, the argument used can be:

  the same type as the formal parameter,

  or,

  any derived class type.

- Static binding is the compile-time determination of which function to call for a particular object based on the type of the formal parameter

- When pass-by-value is used, static binding occurs

Polymorphism – Late Binding

- Allows run-time interpretation of object type for a given class hierarchy
  - Also known as "Late Binding"
- Implemented in C++ using virtual functions

Dynamic Binding

- Is the run-time determination of which function to call for a particular object of a derived class based on the type of the argument
- Declaring a member function to be virtual instructs the compiler to generate code that guarantees dynamic binding
- Dynamic binding requires pass-by-reference

Virtual Functions

- Virtual Functions overcome the problem of run time object determination
- Keyword virtual instructs the compiler to use late binding and delay the object interpretation
- How?
  - Define a virtual function in the base class. The work virtual appears only in the base class
  - If a base class declares a virtual function, it must implement that function, even if the body is empty
  - Virtual function in base class stays virtual in all the derived classes
  - It can be overridden in the derived classes
  - But, a derived class is not required to re-implement a virtual function. If it does not, the base class version is used

Pure Virtual Function

```
class Dog {
    public:
      virtual void bark() = 0;
      void bark() {std::cout<<"??"<<endl;}
};
class 불독 : public Dog {
    public:
      void bark() {std::cout<<"왈왈"<<endl;}
};
class 진돗개 : public Dog {
    public:
      void bark() {std::cout<<"멍멍"<<endl;}
};
class 치와와 : public Dog {
    public:
      void bark() {std::cout<<"깽깽"<<endl;}
};
```

Overriding

Virtual Function

```
class Dog {
    public:
     virtual void bark() = 0;
};
class 불독 : public Dog {
    public:
     void bark() {std::cout<<"왈왈"<<endl;}
};
class 진돗개 : public Dog {
    public:
     void bark() {std::cout<<"멍멍"<<endl;}
};
class 치와와 : public Dog {
    public:
     void bark() {std::cout<<"깽깽"<<endl;}
};
```

CLIENT CODE

```
int main() {
    Dog *dog = new 진돗개;
    dog->bark();

    치와와 Chiwawa;
    dog = &Chiwawa;
    dog->bark();
}
```

Dynamic Binding

OUTPUT

```
멍멍
깽깽
```

Virtual Destructor

```
class Dog {
    public:
     virtual void bark() = 0;
};
class 불독 : public Dog {
    public:
     void bark() {std::cout<<"왈왈"<<endl;}
};
class 진돗개 : public Dog {
    public:
     void bark() {std::cout<<"멍멍"<<endl;}
};
class 치와와 : public Dog {
    public:
     void bark() {std::cout<<"깽깽"<<endl;}
};
```

CLIENT CODE

```
int main() {
    Dog *dog1 = new 진돗개;
    진돗개 *dog2 = new 진돗개;

    delete dog1;
    delete dog2;
}
```

OUTPUT

delete dog1:
- -call only 진돗개's destructor

delete dog2:
- -call class 진돗개's and Dog's destructor

Virtual Destructor

```
class Dog {
    public:
      virtual void bark() = 0;
      ~Dog() {};
};
class 불독 : public Dog {
    public:
      void bark() {std::cout<<"왈왈"<<endl;}
};
class 진돗개 : public Dog {
    public:
      void bark() {std::cout<<"멍멍"<<endl;}
};
class 치와와 : public Dog {
    public:
      void bark() {std::cout<<"깽깽"<<endl;}
};
```

CLIENT CODE

```
int main() {
    Dog *dog1 = new 진돗개;
    진돗개 *dog2 = new 진돗개;

    delete dog1;
    delete dog2;
}
```

Polymorphism Summary

- When you use virtual functions, complier store additional information about the types of object available and created
- Polymorphism is supported at this additional overhead
- Important:
  - Virtual functions work only with pointers/references
  - Not with objects even if the function is virtual
  - If a class declares any virtual methods, the destructor of the class should be declared as virtual as well

[Lab - Practice #1]

- Calculate area of the figure
  - Input
    - Type of figure (triangle | rectangle | pentagon)
    - Points (2D)
  - Output: area of given figure

```
$./figure
Figure: triangle
Point1: 0 0
Point2: 2 0
Point3: 2 2
Area: 2
```