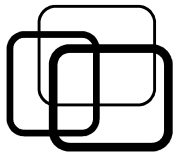
Polymorphism

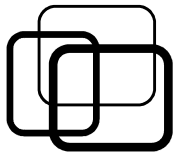
Inst. Nguyễn Minh Huy

Contents



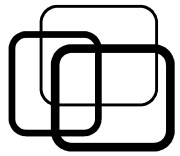
- Basic concepts.
- Virtual function.
- Virtual destructor.

Contents



- **Basic concepts.**
- Virtual function.
- Virtual destructor.

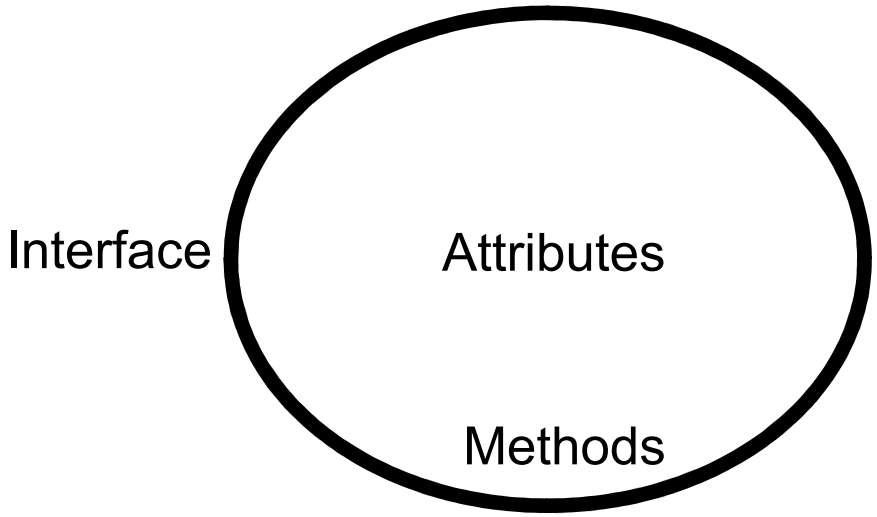
Basic concepts



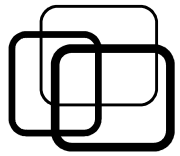
- Interface:
  - Rule of Blackbox:
    - Attributes: **private** to limit access.
    - Methods: **public** to provide functions.

  ➔ Object “talks” to outside through public methods.

  ➔ Public methods declaration ➔ INTERFACE.



Basic concepts



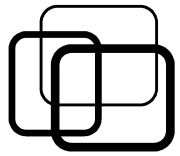
- Interface:
  - Class = **interface** + private + implementation.
  - Interface ~ class prototype (no implementation).
  - Interface defines object communications.

```
class Fraction
{
private:
        int   m_num;
        int   m_den;
public:
        Fraction( int num, int denom );
        Fraction reduce( );
        Fraction inverse( );
};
```

Works with interface Fraction

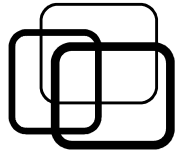
```
void doSomething( Fraction  p ) { }

int main()
{
        Fraction  p1( 1, 2 );
        Fraction  p2( 1, 3 );
        doSomething( p1 );
        doSomething( p2 );
}
```

Basic concepts



- Interface in inheritance:
  - Derived class:
    - Inherits attributes and methods from base class.
    - ➔ Inherits base class interface.
    - ➔ Has same functionalities with base class.
  - Polymorphism in inheritance:
    - Function works with base object…
      - ➔ Can also works with derived object.
    - Pointer to base object…
      - ➔ Can also points to derived object.

Interface



- Example:

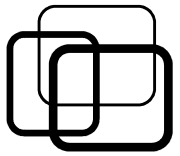
```
class Animal
{
public:
        void talk()  {  cout << “Don’t talk”;  }
};

class Cat: public Animal
{
public:
        void talk()  {  cout << “Meo meo”;  }
};

class Dog: public Animal
{
public:
        void talk()  {  cout << “Gau gau”;  }
};
```

Works with interface Animal

```
void giveSpeech(Animal  p)
{
        p.talk();
}
int main()
{
        Animal   a;
        Cat       c;
        Dog      d;
        giveSpeech( a );
        giveSpeech( c );
        giveSpeech( d );
        Animal  *p;
        p = &a;
        p = &c;
        p = &d;
}
```

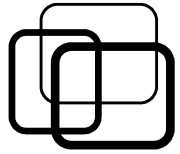
Also works with derives from Animal

Contents



- Basic concepts.
- **Virtual function.**
- Virtual destructor.

Virtual function



- Static binding problem:

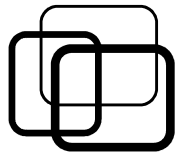
```
class Animal
{
public:
        void talk()  { cout << “Don’t talk”; }
};

class Cat: public Animal
{
public:
        void talk()  { cout << “Meo meo”; }
};

class Dog: public Animal
{
public:
        void talk()  { cout << “Gau gau”; }
};
```

```
void giveSpeech(Animal  p)
{
        p.talk();
}
int main()
{
        Cat       c;
        Dog       d;
        giveSpeech( c );
        giveSpeech( d );

        Animal  *p;
        p = &c;
        p->talk();
        p = &d;
        p->talk();
}
```

Bind to Animal implementation when compile

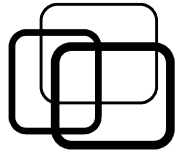
Bind to Animal implementation when compile

Virtual function



- Virtual function concept:
  - Normal function:
    - Function call binds to implementation at compile-time.
      ➔ Static binding.
  - Virtual function:
    - Function call binds to implementation at run-time .
      ➔ Dynamic binding.
      ➔ Implementation depends on run-time object.
  - C++ usage:
    - Declaration: **virtual** <Function prototype>;
    - Called through object pointer or reference.

Virtual function



- Dynamic binding:

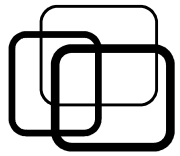
```
class Animal
{
public:
        virtual void talk()  { cout << “Don’t talk”; }
};

class Cat: public Animal
{
public:
        void talk()  { cout << “Meo meo”; }
};

class Dog: public Animal
{
public:
        void talk()  { cout << “Gau gau”; }
};
```

```
void giveSpeech(Animal  *p)
{
        p->talk();
}
int main()
{
        Cat       c;
        Dog       d;
        giveSpeech( &c );
        giveSpeech( &d );

        Animal  *p;
        p = &c;
        p->talk();
        p = &d;
        p->talk();
}
```

implementation depends on run-time object

implementation depends on run-time object

Virtual function

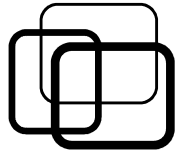


- Pure virtual function:
  - Has declaration only, no implementation.
  - **virtual** <Function prototype> **= 0**.
  - Used for dynamic binding.
  - Derived class provides implementation.
- Abstract class:
  - Class has at least ONE pure virtual function.
  - Cannot create instance, use only for inheritance.

```
class Animal
{
public:
        virtual void talk() = 0;
};
```

Pure virtual function, has no implementation!!

Virtual function



- Example:

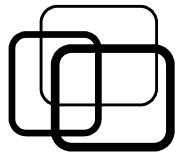
```
class Animal
{
public:
        virtual void talk() = 0;
};

class Cat: public Animal
{
public:
        void talk()  {  cout << “Meo meo”;  }
};

class Dog: public Animal
{
public:
        void talk()  {  cout << “Gau gau”;  }
};
```

Abstract class

```
void giveSpeech(Animal  *p)
{
        p->talk();
}
int main()
{
        Cat       c;
        Dog       d;
        giveSpeech( &c );
        giveSpeech( &d );

        Animal  *p;
        p = new Animal; // Wrong
        p = new Cat;      // Right
        p->talk();
}
```

implementation depends on run-time object

Virtual function



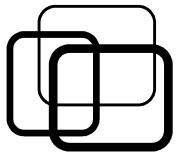
- Polymorphism meaning:
  - Communicate through interface.
  - Implementation can be changed at run-time.

    ➔ Abstract programs.

```
void giveSpeech( Animal *p )
{
        p->talk( );
}
```

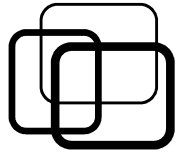
```
void giveSpeech( int type )
{
        if ( type == 0 )
        {
                Cat  c;
                c.talk( );
        }
        else if ( type == 1 )
        {
                Dog  d;
                d.talk( );
        }
}
```

Contents



- Basic concepts.
- Virtual function.
- **Virtual destructor.**

Virtual destructor



- Example:

```
class Teacher
{
private:
        char       *m_name;
public:
        ~Teacher()  {  delete m_name;  }
};
class HRTeacher : public Teacher
{
private:
        char       *m_classroom;
public:
        ~HRTeacher()  {  delete m_classroom;  }
};
```

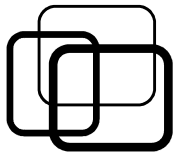
```
int main()
{
        HRTeacher  *p1 = new HRTeacher;
        delete  p1;

        Teacher  *p2 = new HRTeacher;
        delete  p2;
}
```

~HRTeacher()
~Teacher()

~Teacher()

Why destructor calls are different??

Virtual destructor



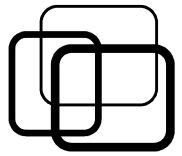
- Dr. Guru advises:
  - **Destructor should be virtual function.**

    ➔ Dynamic binding, can be used with polymorphism.

```
class Teacher
{
public:
        virtual ~Teacher() { delete m_name; }
};
class HRTeacher : public Teacher
{
public:
        ~HRTeacher()  {  delete m_classroom;  }
};
Teacher  *p3 = new HRTeacher;
delete  p3;
```

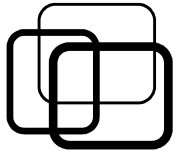
~HRTeacher()
~Teacher()

Summary



- Basic concepts:
  - Interface ~ class prototype (public declaration).
  - Derived class has base class interface.
  - Polymorphism: derived object can disguise base object.
- Virtual function:
  - Implementation bound at run-time.
  - Use keyword **“virtual”**.
- Virtual destructor:
  - Destructor should be virtual function.

Practice



- Practice 9.1:

```
class A {
public:
[yyy] void f1() { cout << “Good morning.\n”; f2(); }
[zzz] void f2() { cout << “Good afternoon.\n”; }
};
class B: public A {
public:
        void f1() { cout << “Good evening.\n”; f2(); }
        void f2() { cout << “Good night.\n”; }
};
int main()
{
        A   *p = new B;
        p->f1();
}
```

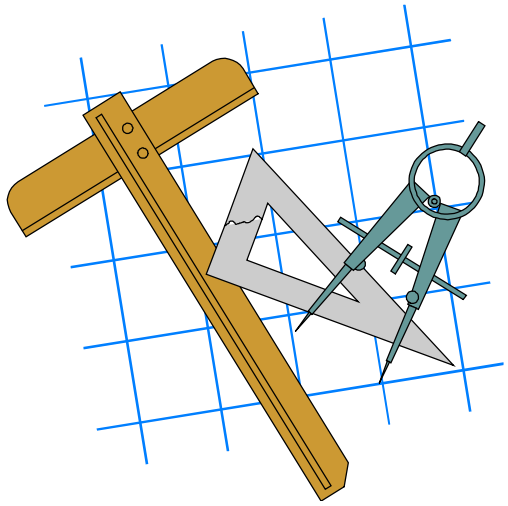
What are displayed on screen for each of the following cases:

a) [yyy] empty, [zzz] empty.

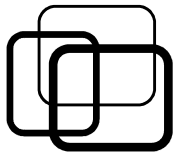
b) [yyy] empty, [zzz] virtual.

c) [yyy] virtual, [zzz] empty.

d) [yyy] virtual, [zzz] virtual.



Practice



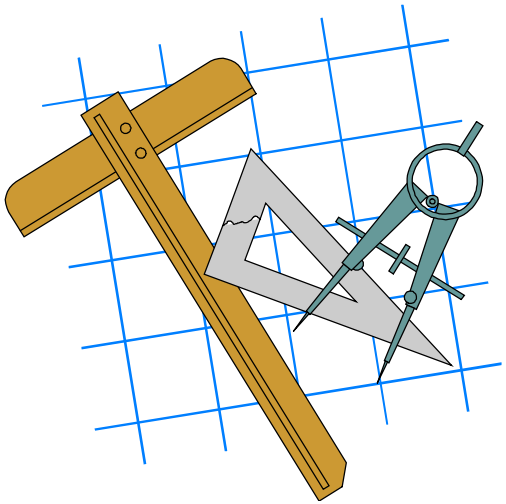
- ## Practice 9.2:

  There are 2 types of shapes:

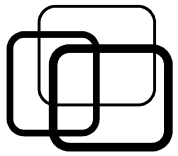
  - **Triangle**: represented by 3 points.
  - **Rectangle**: represented by 2 points (top-left and bottom-right).

  a) Write **printShapes**(vector<Triangle> v1, vector<Rectangle> v2) to print information of all triangles and rectangles in the input lists (use Encapsulation).

  b) Add new type of shape **Circle** (represented by center and radius). Update printShapes in a) to due with the added type.
  How about using Polymorphism?



Practice



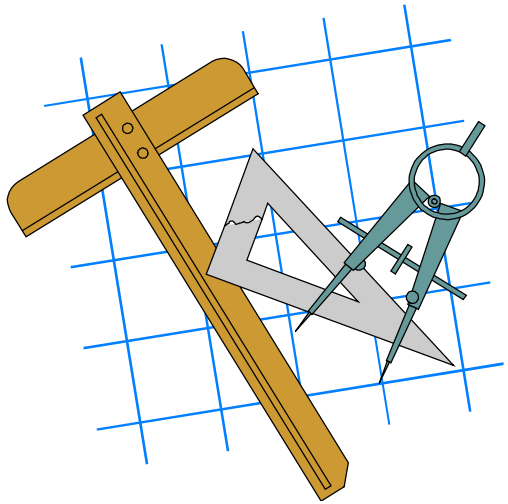
- ## Practice 9.3:

  The table below tells max speeds of some animals:

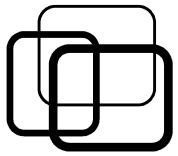
Animal	Speed
Cheetah	100km/h
Antelope	80km/h
Lion	70km/h
Dog	60km/h
Human	30km/h

Write function to takes 2 animals from the table as arguments then tells which animal wins the race.

Add the horse (60km/h) to the table, what changes are made in the code?



Practice



- Practice 9.4 (*):

Given class **Line** and **Rect** as follow:

```
class Line
{
private:
        Point  m_p1;
        Point  m_p2;
public:
        Line(Point, Point);
        void drawLine();
};
```

```
class Rect
{
private:
        Point  m_p1;
        Point  m_p2;
public:
        Rect(Point, Point);
        void drawRect();
};
```

Write a function to draw lines and rectangles from a list.

Requirements:

- Use (no update) class Line and Rect to draw.
- The function need to be unchanged when add another type of shape.