

EEE102

C++ Programming and Software Engineering II

## Lecture 8 Polymorphism

**Dr. Rui Lin / Qing Liu**

**Rui.Lin / Qing.Liu@xjtlu.edu.cn**

**Room EE512 / EE516**

**Office hour: 2-4pm, Tuesday & Wednesday**

**/ Monday & Wednesday**



# Outline

- Pointers to objects
  - Pointers to objects
  - Pointers to derived classes
- Polymorphism
  - Methods Overlapping
  - Introduction to Polymorphism
    - Static Binding
    - Dynamic Binding
  - Virtual Methods
  - Pure Virtual Methods
  - Virtual destructor



# 1.1 Pointers to objects

- A pointer can point to an object created by a class.

– Example:

```
complexClass cNum1;  
complexClass *ptr1;  
ptr1 = &cNum1;
```

– Call methods in two ways:

- dot operator (object):

```
cNum1.show();  
cNum1.set(5,10);
```

- arrow operator (object pointer):

```
ptr1->show();  
ptr1->set(5,10);
```



# 1.1 Pointers to objects

- Dynamic memory allocation for object

- Example:

```
complexClass *ptr1 = new complexClass;  
ptr1-> show();  
ptr1-> set(5,10);  
delete ptr1;
```

- Dynamic memory allocation for objects array

- Example:

```
int N=5;  
complexClass *ptr2 = new complexClass[N];  
ptr2[0].show();  
*(ptr2+1)->set(5,10);  
delete [] ptr2;
```



## 1.2 Pointers to derived classes (I)

- Pointers can be used to point not only base class objects but also objects of derived classes.
  - Pointers to objects of a base class are type-compatible with pointers to objects of a derived class.
  - Therefore, a single pointer variable can be made to point to objects belonging to different classes.

```
class clA
{public:
    int a;
    void show() {cout<<a;}
};

class clB: public clA
{public:
    int b;
    void show() {cout<<a<<b;}
};
```

```
clA *ptr1, obA;
clB obB;
ptr1 = &obA;
ptr1 = &obB;
```

```
clA *ptr2 = new clA;
clA *ptr3 = new clB;
clB *ptr4 = new clA;
```

**X**

## 1.2 Pointers to derived classes (II)

- Problems:

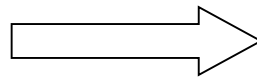
- 1. Pointer declared from derived class cannot point to object of base class;

```
clA p1;  
clB *ptr2 = &p1; X
```

```
clB p2;  
clA *ptr1 = &p2; ✓
```

- 2. Using **ptr1**, we can access only the members which are inherited from base class, but not the members originally defined in derived class.

```
clB p2;  
clA *ptr1 = &p2;
```



```
ptr1->a = 1; ✓  
ptr1->b = 1; X
```




## 1.2 Pointers to derived classes (III)

- Problems:
  - 3. In case a method of derived class has the same name as the method of base class, then any reference to that member by **ptr1** will always access the base class member.

```
class clA
{public:
    int a;
    void show() {cout<<a;}
};

class clB: public clA
{public:
    int b;
    void show() {cout<<a<<b;}
};
```

```
int main()
{
    clB p2;
    clA *ptr1 = &p2;
    ptr1-> show();
    return 0;
};
```



## 2.1 Methods Overlapping

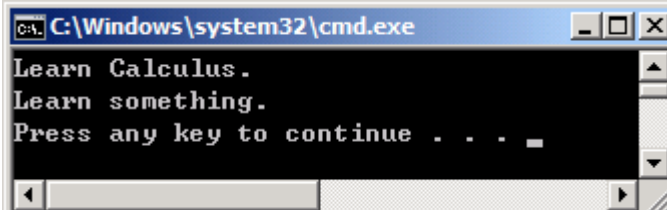
- Function member in base class and subclass can have same name, same parameter list.

Not overloading !!!

```
class student
{public:      .....
    void study()
    {cout<<"Learn something."<<endl;}
protected: .....
};

class undergraduate: public student
{public:      .....
    void study()
    {cout<<"Learn Calculus."<<endl;}
protected: .....
};
```

```
int main()
{
    undergraduate s1;
    s1.study();
    student s2;
    s2.study();
    return 0;
}
```



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window shows the output of the program: "Learn Calculus." on the first line, "Learn something." on the second line, and "Press any key to continue . . . ." on the third line. The cursor is positioned at the end of the third line.





## 2.2 Introduction to Polymorphism

- Polymorphism is one of the crucial features of OOP
  - Polymorphism means a function in the derived class can have the same name as in the base class but does different things.
  - “One name, multiple forms”
- Types of polymorphism:
  - Compile time polymorphism
  - Run time polymorphism

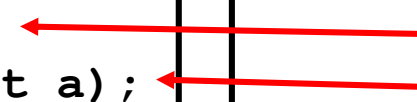


## 2.2.1 Static Binding

- Overloaded functions and operators
  - Appropriate overloaded function are selected for invoking by matching arguments list;
  - Known to compiler at the compilation stage;
  - Called “early binding” “static binding” “compile time polymorphism”

```
class clA
{
    int x;
public:
    void show ();
    void show (int a);
};
```

```
int main()
{
    clA ob1;
    clA *ptr = &ob1;
    ptr->show();
    ptr->show(5);
    return 0;
};
```

Two red arrows originate from the `ptr->show();` and `ptr->show(5);` lines in the `main()` function box. The first arrow points to the `void show ();` line in the `clA` class box. The second arrow points to the `void show (int a);` line in the `clA` class box. This illustrates how the compiler statically binds the function calls to the specific overloaded methods based on the arguments provided.

# The “methods overlap” is one kind of polymorphism.

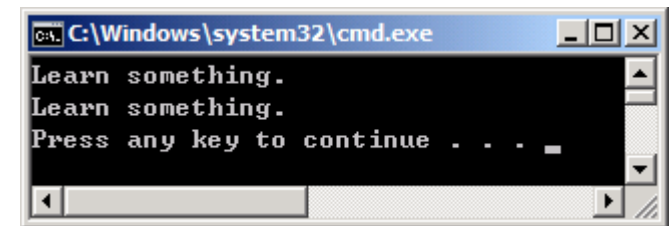
- Static binding:
  - to call the function of the base class from an object of the derived class
  - determined in compile time

```
class student
{public:
    .....
    void study()
    {cout<<"Learn something."<<endl;}
protected: .....
};

class undergraduate: public student
{public:
    .....
    void study()
    {cout<<"Learn Calculus."<<endl;}
protected: .....
};
```

## Static Binding

```
int main()
{
    undergraduate s1;
    s1.student::study();
    student s2;
    s2.study();
    return 0;
}
```

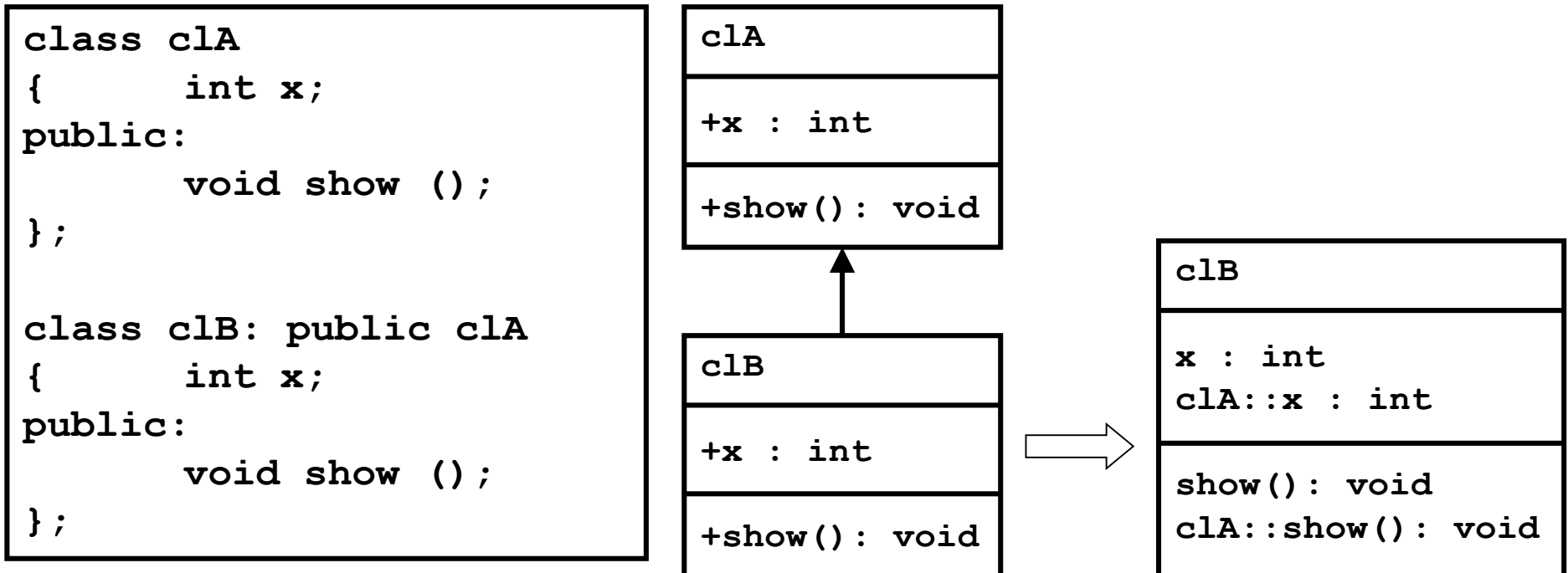


A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window shows the output of the program: "Learn something.", "Learn something.", and "Press any key to continue . . .". The cursor is positioned after the last line of output.



## 2.2.2 Dynamic Binding

- Dynamic Binding
  - Which function should be called will be decided during the execution of a programme.
  - Called “late binding” “run time polymorphism”



## 2.2.2 Dynamic Binding

```
clA ob1, *ptr;  
// ptr is a pointer pointing to the object of class clA  
  
clB ob2;  
  
//A pointer of a base class type can be used to point to a derived class  
if (.....)  
    ptr = &ob1;    // pointing to an object of the base class  
else  
    ptr = &ob2;.    // pointing to an object of the derived class  
  
ptr->show();
```

*Actually, since **ptr** is declared as a pointer for **clA** class, the method **show()** of **clA** is always called. Not **dynamic** !*

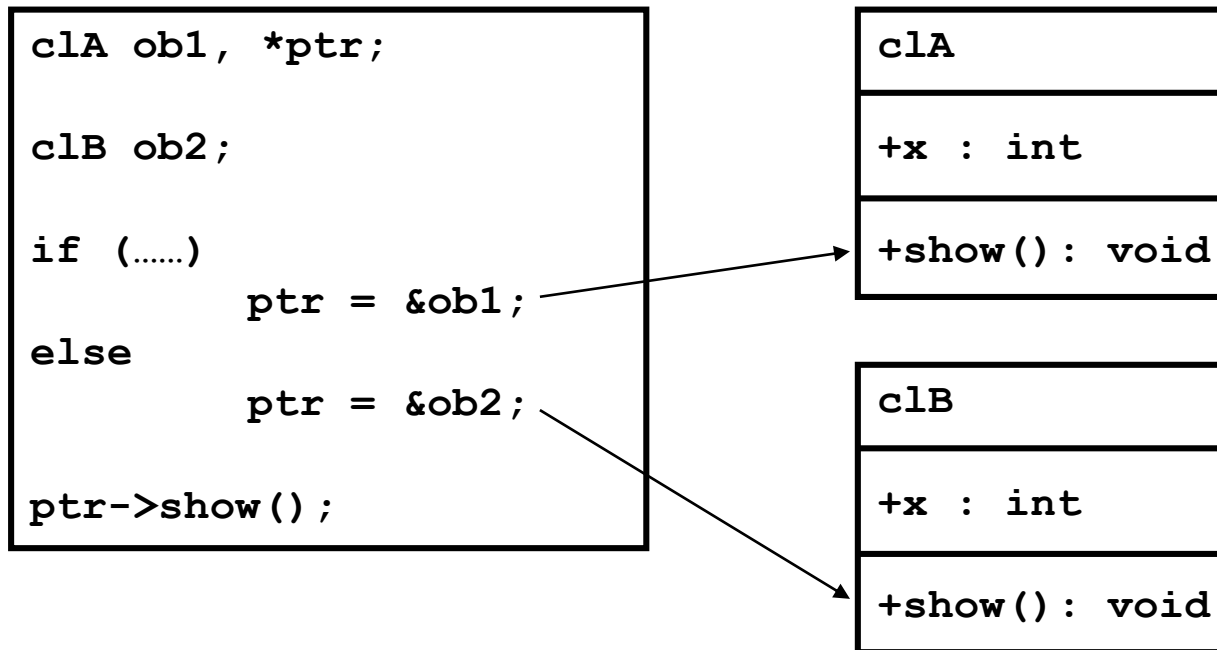
**Which function will be called ?**

**Dynamic binding ---- which function should be called will be decided during the execution of a programme.**



## 2.3 Virtual Methods

- Solution: Using “virtual methods” *In the base class*
  - add keyword “**virtual**” in front of the methods declaration.
  - Then the program will find out which methods should be called according to what object the pointer is pointing to.



## 2.3.1 Example

```
class student
{public: .....
    virtual void study()
    {cout<< "Learn something."<<endl;}
protected: .....
};

class undergraduate: public student
{public: .....
    void study()
    {cout<< "Learn Calculus."<<endl;}
protected: .....
};
```

```
int main()
{
    cout<<"Choose: \n";
    cout<<"1 for student; \n";
    cout<<"2 for undergraduate: \n";
    cin>>choice;
    student st1, *pst;
    undergraduate ust2;
    if (choice==1)
        pst=&st1;
    else
        pst=&ust2;
    pst->study();
    return 0;
}
```

C:\Windows\system32\cmd.exe

```
Choose:
1 for student;
2 for undergraduate:
1
Learn something.
Press any key to continue . . .
```

C:\Windows\system32\cmd.exe

```
Choose:
1 for student;
2 for undergraduate:
2
Learn Calculus.
Press any key to continue . . .
```

## 2.3.2 Rules for Virtual Methods

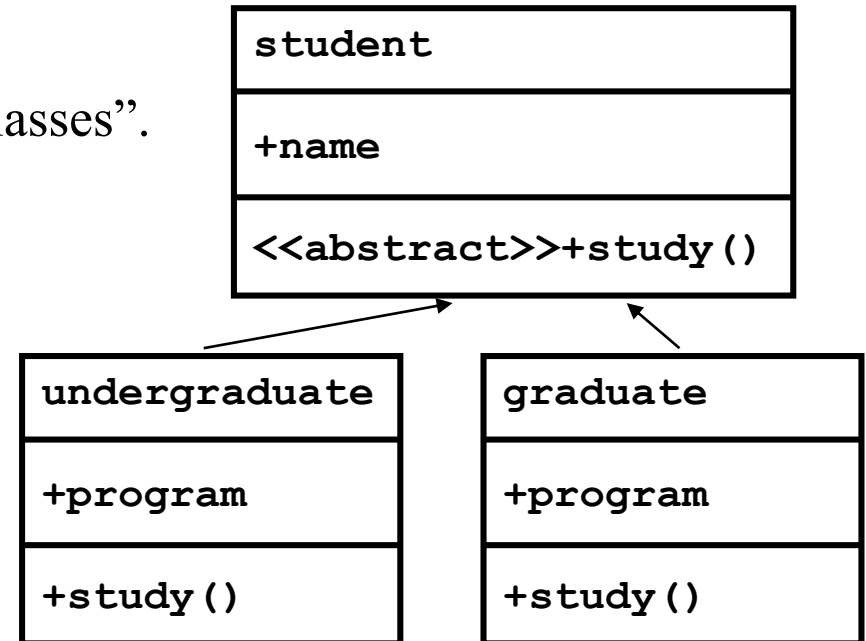
- Basic rules for “virtual methods”
  - The virtual functions must be members of some class;
  - They are accessed by using object pointers;
  - A virtual function can be a friend of another class;
  - A virtual function in a base class must be defined, even though it may not be used;
    - If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class.
  - The prototypes of the base class version of a virtual function and all the derived class versions must be identical;
    - meaning “same name, same parameter list”
    - If different, they will be considered as “function overloading”





## 2.4 Pure Virtual Methods

- If the virtual function defined in base class doesn't perform any task, but only serves as a *placeholder*, it is a “do-nothing” function.
  - Such functions are called “pure virtual functions”.
  - Syntax:  
**virtual void study()=0;**
- A class containing pure virtual functions cannot be used to declare any objects of its own.
  - Such classes are called “abstract base classes”.
  - The main objective of an abstract base class is
    - to provide some traits to the derived classes
    - to create a base pointer required for achieving run time polymorphism.



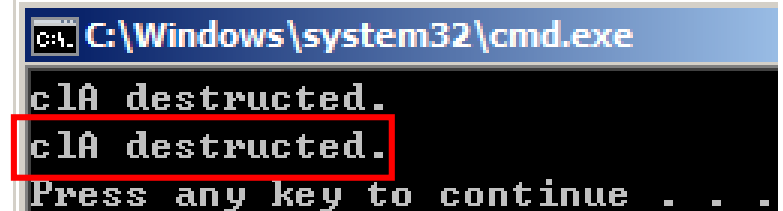
## 2.5 Virtual destructor

- It is a good policy to always make destructors virtual.
  - Why?
  - Example:

```
class clA
{public:
    ~clA()
    {cout<<"clA destructed.\n";}
};

class clB: public clA
{public:
    ~clB()
    {cout<<"clB destructed.\n";}
};
```

```
int main()
{
    clA *ptr1 = new clA;
    delete ptr1;
    clA *ptr2 = new clB;
    delete ptr2;
    return 0;
};
```



```
C:\Windows\system32\cmd.exe
clA destructed.
clA destructed.
Press any key to continue . . .
```



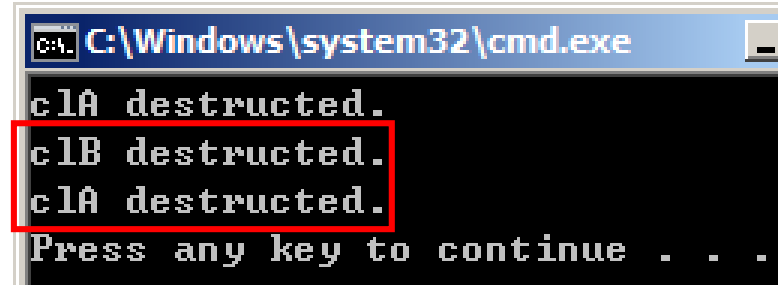
## 2.5 Virtual destructor

- It is a good policy to always make destructors virtual.
  - Why?
  - Example:

```
class clA
{public:
    virtual ~clA()
    {cout<<"clA destructed.\n";}
};

class clB: public clA
{public:
    ~clB()
    {cout<<"clB destructed.\n";}
};
```

```
int main()
{
    clA *ptr1 = new clA;
    delete ptr1;
    clA *ptr2 = new clB;
    delete ptr2;
    return 0;
};
```



```
C:\Windows\system32\cmd.exe
clA destructed.
clB destructed.
clA destructed.
Press any key to continue . . .
```



### 3. Example of Polymorphism

- Attack()



Swordsman



Archer



Magician

### 3. Example of Polymorphism

- A character is try to perform attack action, but the attack for all 3 jobs are different:
  - swordsman :: chop, damage should be determined by player's AP and enemy's DP
  - Archer :: shoot, damage should be determined by player's speed and enemy's DP
  - Magician :: fire ball, damage should be determined by player's intelligence
- Since which job the user is using will be determined according to user's choice, in execution stage. Therefore, the run-time polymorphism is needed here.



### 3. Example of Polymorphism

```
class player
{
public:
    virtual void attack();
};
```

```
class swordsman : public player
{
public:
    void attack(); // chop
};
```

```
class magician: public player
{
public:
    void attack(); // fireBall
};
```

```
class archer: public player
{
public:
    void attack(); // shoot
};
```



# Summary

- Modern object-oriented programming (OOP) technique provide 3 capabilities:
  - *Encapsulation*: is the process of combining data and functions into a single unit called class.
  - *Inheritance*: a means of specifying hierarchical relationships between classes.
  - *Polymorphism*: is the ability to use an operator or function in different ways.

