# 15. Polymorphism & Virtual Functions

**Hu Sikang**
***skhu@163.com***

# Contents

- **Virtual Function**
- **Use Virtual Function**
- **Function Call Binding**
- **Abstract Class**
- **Pure Virtual Destructor**
- **Downcasting**

# 15.1 Virtual Functions

## CASE 1: Why use virtual function?

```
#include <iostream>
using namespace std;
class Instrument {
public:
        void play( ) const
        {  cout << "Instrument::play()" << endl; }
};
class Wind : public Instrument {
public:
        void play( ) const
        {  cout << "Wind::play()" << endl;    }
};
class Stringed: public Instrument {
public:
        void play( ) const
        {  cout << " Stringed::play()" << endl;  }
};
```

```
void tune(const Instrument& i)
{
    i.play( );
}

int main( )
{
    Wind flute;
    tune(flute);

    Stringed guitar;
    tune(guitar);

    return 0;
}
```

# 15.1 Virtual Functions(1)

```
#include <iostream>
using namespace std;
class Instrument {
public:
        virtual void play( ) const
        {  cout << "Instrument::play( )" << endl;  }
};
void tune(const Instrument& instru)   {   instru.play( );  }
```

**The keyword *virtual* indicates that *play( )* can act as an interface to the *play( )* function defined in this class and the *play( )* functions implementations in classes derived from it.**

# 15.1 Virtual Functions

## CASE 1: Why use virtual function?

```
#include <iostream>
using namespace std;
class Instrument {
public:
        virtual void play( ) const
        {  cout << "Instrument::play()" << endl; }
};
class Wind : public Instrument {
public:
        virtual void play( ) const
        {  cout << "Wind::play()" << endl;    }
};
class Stringed: public Instrument {
public:
        void play( ) const   // virtual can be omitted
        {  cout << " Stringed::play()" << endl;  }
};
```

```
void tune(const Instrument& i)
{
        i.play( );
}

int main( )
{
        Wind flute;
        tune(flute);

        Stringed guitar;
        tune(guitar);

        return 0;
}
```

# 15.1 Virtual Functions(2)

**CASE 2: Why use virtual function?**

```
#include <iostream>
using namespace std;

class Instrument {
public:
        void tune()  {   play( );   }
        void play()  const
        {  cout << "Instrument::play()" << endl;  }
};
```

virtual void play() const;

```
class Wind : public Instrument
{
public:
        void play() const
        {  cout << "Wind::play()" << endl;    }
};
```

```
class Stringed : public Instrument
{
public:
      void play() const
      {
          cout << " Stringed ::play()";
      }
};

int main( ) {
      Wind flute;
      flute.tune();

      Stringed guitar;
      guitar.tune();
      return 0;
}
```

# 15.1 Virtual Functions(3)

**[1] There must be the same function definition when overloading the virtual function. It includes *same returning type, same function name, same arguments number, same arguments* sequence and *same arguments type*.**

**[2] The virtual function must be a member function.**

**[3] The friend function cannot be defined as a virtual function.**

**[4] Destructor can be defined as a virtual function, but constructor cannot.**

# 15.1 Virtual Functions(4)

Practice 1

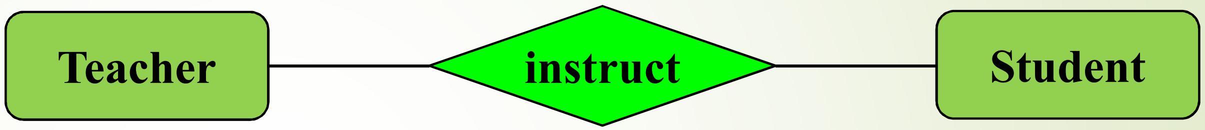
```
class  base
{ public :
      virtual  void  vf1 ( ) ;
      virtual  void  vf2 ( ) ;
      virtual  void  vf3 ( ) ;
      void  f ( ) ;
 } ;
int main ( ) {
   derived   d ;
   base   * bp = & d ;
   bp -> vf1 ( ) ;          // call derived :: vf1 ( )
   bp -> vf2 ( ) ;          // call base :: vf2 ( )
   bp -> f ( ) ;            // call base :: f ( )
   return 0;
} ;
```

```
class  derived : public  base
{
public :
      void  vf1 ( ) ;       // virtual function
      // overloading, but not a virtual
      void  vf2 ( int ) ;
      char  vf3 ( ) ;       // error
      void f ( ) ;          // Not overload virtual
 } ;
```

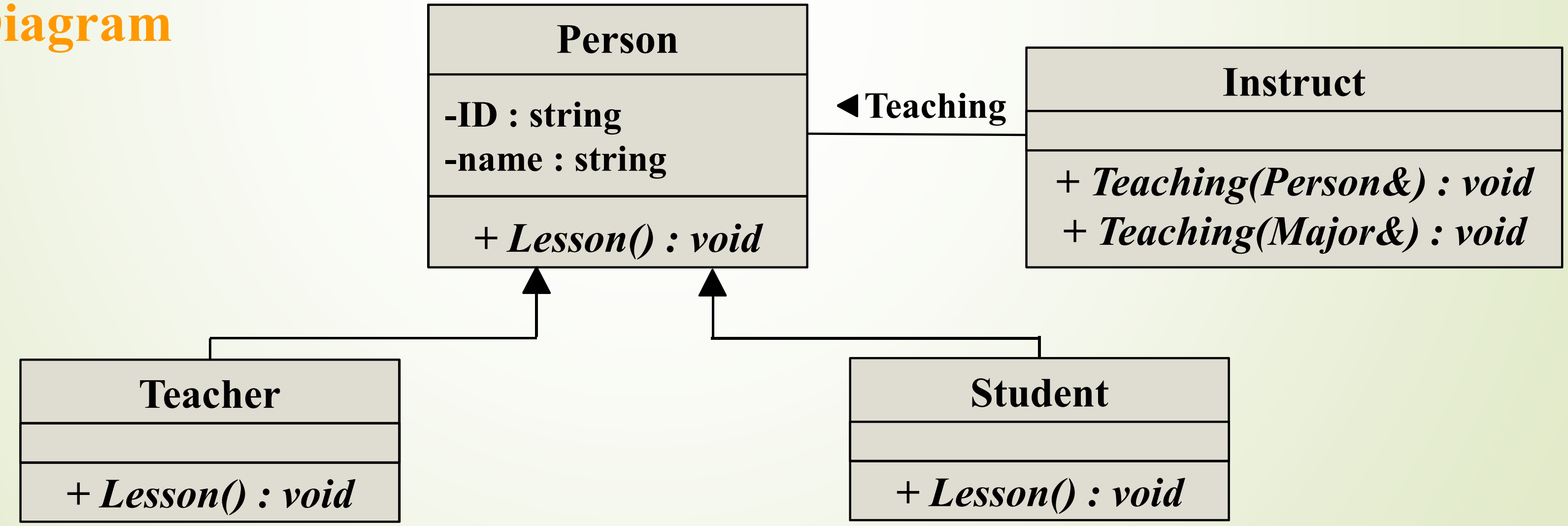
# 15.2 Use Virtual Functions

## Practice 2

ER Model




Class Diagram



# 15.2 Use Virtual Functions

## Practice 2

```
#include <iostream>
using namespace std;
class Person {
private:   string ID, name;
public:
      virtual void Lesson( )  {
            cout << "Person has a lesson ." << endl;
      }
};
class Teacher : public Person  {
public:
      virtual void Lesson( )  override  {
            cout << "Teacher teaches." << endl;
      }
};
class Student : public Person  {
public:
      virtual void Lesson( ) {
            cout << "Student does exercises. " << endl;
      }
};
```

```
class Instruct
{
public:
     void Teaching(Person& p)   {
            p.Lesson( );
     }
     // instruct graduate to write thesis
     void Teaching(Major& s)   {
            s.Thesis( );
     }
};

int main( ) {
      Teacher teacher;
      Student student;

      Instruct instruct;
      instruct.Teaching(teacher);
      instruct.Teaching(student);
      return 0;
}
```

# 15.3 Virtual Destructors

- Calling the wrong destructor could be disastrous, particularly when it contains a *delete* statement.
- Destructors are not inherited.
- Constructors inherited? No.

# 15.3 Virtual Destructors(1)

```
#include <iostream>
using namespace std;
class  Item  {
 public:
            Item()    { id = 0;  }
            ~Item()  { cout <<"Item deleted"<<endl;}
 private:  int id;
};

 class BookItem : public Item {
 public:
     BookItem()  {    title = new char [50];    }
     ~BookItem()
     {
          cout <<"BookItem deleted"<<endl;
          if (title != nullptr)     delete[ ] title;
     }
 private:       char * title;
};
```

```
int main()
{
        Item * p;
        p = new Item();
        delete p;

        p = new BookItem();
        delete p;

        return 0;
}
```

C:\Windows\system32\cmd.exe

```
Item deleted
Item deleted
```

# 15.3 Virtual Destructors(2)

```
#include <iostream>
using namespace std;
class  Item  {
 public:
              Item()    { id = 0;  }
              virtual ~Item()  {  cout <<"Item deleted"<<endl; }
 private:  int id;
};

 class BookItem : public Item {
 public:
      BookItem()  {    title = new char [50];    }
    ~BookItem()
     {
            cout <<"BookItem deleted"<<endl;
            if (title != nullptr)     delete[ ] title;
      }
 private:      char * title;
};
```

```
int main()
{
        Item * p;
        p = new Item();
        delete p;

        p = new BookItem();
        delete p;

        return 0;
}
```

Microsoft Visual Studio 调试控制台

```
Item deleted
BookItem deleted
Item deleted
```

# 15.4 Function Call Binding

Connecting a function call to a function body is called *binding*. When binding is performed before the program is run(by the compiler and linker), it's called *early binding* or *static binding*.

```
#include <iostream>
using namespace std;

class Person {
private:  string ID, name;
public:
        void Lesson( )
        {
            cout << "Person has a lesson ." << endl;
        }
};

class Teacher : public Person  {
public:
        void Lesson( ) {
             cout << "Teacher is teaching." << endl;
        }
};
```

```
class Student : public Person
{
public:
        void Lesson( )
       {
            cout << "Student is listening. ";
        }
};
class Instruct
{
public:
      void Teaching(Person& p)
       {
           p.Lesson( );
       }
};
```

No Polymorphism

```
int main( )  {
        Teacher teacher;
        Student student;
        Instruct instr;
        instr. Teaching(teacher);
        teach. Teaching(student);
        return 0;
}
```

# 15.4.1 Function Call Binding

**The solution is called *late binding*, which means the binding occurs at runtime, based on the type of the object. Late binding is also called *dynamic binding* or *runtime binding*.**

```
#include <iostream>
using namespace std;

class Person {
private:  string ID, name;
public:
      virtual void Lesson( )
      {
           cout << "Person has a lesson ." << endl;
      }
};

class Teacher : public Person  {
public:
      virtual void Lesson( ) override  {
           cout << "Teacher is teaching." << endl;
      }
};
```

```
class Student : public Person
{
public:
      void Lesson( )
     {
          cout << "Student is listening. ";
      }
};
class Instruct
{
public:
     void Teaching(Person& p)
      {
          p.Lesson( );
      }
};
```

**Polymorphism**

```
int main( )  {
      Teacher teacher;
      Student student;
      Instruct instr;
      instr. Teaching(teacher);
      teach. Teaching(student);
      return 0;
}
```

# 15.4.1 Function Call Binding

*To simulate printer in Word:*

**Base Class:** ***PRINTER*****,** **Derived Class:** ***MyPrinter***

```
class PRINTER
{
public:
      // printer driver
      virtual int print(CObject* pObj);
};
```

```
class MyPrinter : public PRINTER
{
public:
      // override printer driver
      virtual int print(CObject* pObj);
      // register my printer in the registry of OS
      bool RegisterMyPrinter();
};
```

```
CHandle API_PRINTER
(PRINTER& p, CObject* pObj)
{
      if (Find_In_Register(p) &&
          Default_Printer(p))
      {
          Call  p.print(pObj);
      }
      else
          cout << "No printer";
}
```

# 15.4.2 How C++ implements late binding

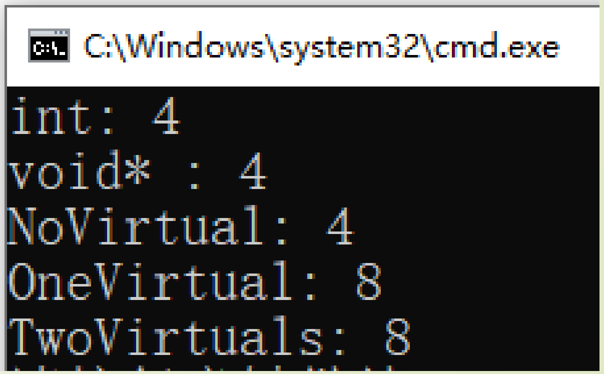
```
#include <iostream>
using namespace std;
class NoVirtual {
   int a;
public:
   void x() const { }
   int i() const { return 1; }
};

class OneVirtual {
   int b;
public:
   virtual void x() const { }
   int i() const { return 1; }
};

class TwoVirtuals {
   int c;
public:
   virtual void x() const { }
   virtual int i() const { return 1; }
};
```

*What does compiler do for us (1)?*

```
int main() {
   cout << "int: " << sizeof(int) << endl;
   cout << "void* : " << sizeof(void*) << endl;
   cout << "NoVirtual: " << sizeof(NoVirtual) << endl;
   cout << "OneVirtual: " << sizeof(OneVirtual) << endl;
   cout << "TwoVirtuals: "<< sizeof(TwoVirtuals) << endl;
   return 0;
}
```



# 15.4.2 How C++ implements late binding

*What does compiler do for us (2)?*

```
#include <iostream>
using namespace std;

class NoVirtual {
public:
    int a;
    void x() const {}
    int i() const { return 1; }
};

class OneVirtual {
public:
    int b;
    virtual void x() const {}
    int i() const { return 1; }
};
```

```
int main() {
    NoVirtual nov;
    OneVirtual onev;

    cout << "NoVirtual: " << &nov << endl;
    cout << "NoVirtual: a " << &nov.a << endl;

    cout << "OneVirtual: " << &onev << endl;
    cout << "OneVirtual: b " << &onev.b << endl;
    return 0;
}
```

C:\Windows\system32\cmd.exe

```
NoVirtual: 0066FB80
NoVirtual: a 0066FB80
OneVirtual: 0066FB70
OneVirtual: b 0066FB74
```

# 15.4.2 How C++ implements late binding

***What does compiler do for us (3)?***

**Here is a piece of source code:** ***i.adjust(1);***

| | | |
|---|---|---|
| **mov** | **esi,esp** | **// balance stack** |
| **push** | **1** | **// argument of *adjust*** |
| **mov** | **eax,dword ptr [i]** | **// *this*, address of *i*** |
| **mov** | **edx,dword ptr [eax]** | **// fetches the word that *si* points to** |
| | | **// it's the VPTR(pointer of virtual table)** |
| **mov** | **ecx,dword ptr [i]** | **// *this*, address of *i*** |
| **mov** | **eax,dword ptr [edx+8]** | **// address of adjust. *Why add 8?*** |
| **call** | **eax** | |
| **cmp** | **esi,esp** | **// balance stack** |

# 15.5 Variant return type

**[1] There must be the same function definition when overloading the virtual function. It includes *same returning type*, *same function name*, *same arguments number*, *same arguments sequence* and *same arguments type*.**

**[2] The virtual function must be a member function.**

**[3] The friend function cannot be defined as a virtual function.**

**If we are *returning* *a pointer or a reference of an object* to a base class, then the overridden version of the function may *returning* *a pointer or a reference of an object* to a class derived from what the base returns.**

# 15.6 Abstract Classes

A ***pure virtual function*** **is a virtual function that contains a pure-specifier, designated by the** ***"=0"*****. It's used to be defined as a interface of derived class.**

```
class  Number        // Abstract class
{
 public :
        Number ( int  i )  { val = i ; }
        virtual   void   Show ( ) = 0 ;    // pure virtual function
protected :
        int   val ;
} ;
```

# Number Abstract

```
#include < iostream.h >
class  Number
{ public :
      Number ( int  i )  { val = i ; }
      virtual   void   Show ( ) = 0 ;
  protected :   int   val ;
} ;
class   Hextype  :  public  Number
{ public :
      Hextype ( int  i ) : Number ( i ) { }
      void  Show ( ) {   cout << hex << val;   }
} ;
class  Dectype  :  public  Number
{ public :
      Dectype ( int  i ) : Number ( i ) { }
      void  Show ( ) {   cout << dec << val;  }
} ;
```

```
void   Show (const Number& n)
{
      n. Show ( ) ;
 }

int main ( )
{
      Dectype  d ( 50 ) ;
      Show( d ) ;    // d . Show ( ) ;

      Hextype h ( 16 ) ;
      Show( h ) ;    // h . Show ( ) ;
      return 0;
}
```

# 15.6 Abstract Classes

- An abstract class is a class that can only be a base class for other classes.
- Abstract classes represent *concepts* for which objects cannot exist.
- Abstract class couldn't define instances.
- The derived classes of abstract class are used to instantiate objects.

# 15.6 Pure virtual destructor

**In common sense, we don't give the source code for pure virtual function. But in the special, it's possible to provide a definition for a pure virtual function in the base class.** ***There may be a common piece of code that we want some or all of the derived class definitions to call rather than duplicating that code in every function*****.**

```
#include <iostream>
using namespace std;
class Pet  {
public:   virtual ~Pet() = 0;
};
// Don't implement in the class
Pet::~Pet( ) {  cout << "~Pet()" << endl; }

class Dog: public Pet {
public:   ~Dog() {   cout << "~Dog()" << endl;   }
};
```

```
int main()
{
        // Upcase
          Pet *p = new Dog();

        // Virtual destructor call
          delete p;
          return 0;
}
```

# 15.7 Downcasting

C++ provides a special explicit cast called *dynamic_cast* that is a *type-safe* ***downcast*** operation. When we use *dynamic_cast* to try to cast down to a particular type, the return value will be a ***pointer*** to the desired type only if the cast is proper and successful, otherwise it will return ***zero***.

# 15.7 Downcasting

```
#include <iostream>
using namespace std;
class Pet  {   public:  virtual ~Pet() { }    };
class Dog : public Pet {   };
class Cat  : public Pet {   };
int main( )
{
        Pet *b = new Cat();   // Upcase
        // Try to cast it to Dog*
        Dog* d1 = dynamic_cast<Dog*>(b);    // What would happen to Dog* d1 = (Dog*)b;
        // Try to cast it to Cat*
        Cat* d2 = dynamic_cast<Cat*>(b);
        cout << "d1 = " << d1 << endl;
        cout << "d2 = " << d2 << endl;
        delete b;     // call base destructor automatically
        return 0;
}
```

```
C:\Windows\system32\cmd.exe
d1 = 00000000
d2 = 011F0500
```