Polymorphism

C++ Object Oriented Programming
Pei-yih Ting
NTOU CS

Contents

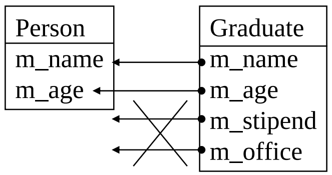
Assignment to Base Class Object

- Assume Graduate is derived from Person
  Assignment from derived class object to base class object is legal though unusual

```
Person      person("Joe", 19);
Graduate graduate("Michael", 24, 6000, "8899 Storkes");
person.display();
person = graduate;
person.display();
Person      person2 = graduate;
person2.display();
```

Output
Joe is 19 years old.
Michael is 24 years old.
Michael is 24 years old.

- What happened:
  1. A derived object, by definition, contains everything the base class has plus some extra elements.
  2. The extra elements are lost in the assignment.



- If the **base class** has implemented the assignment operator or the copy ctor, they will be called.

Assignment to Derived Class Object

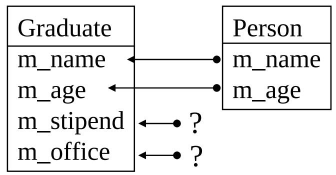
- Assignment from base class object to derived class object is **illegal**

```
graduate = person;
Graduate graduate2 = person;
```

error C2679: binary '=' : no operator defined which takes a right-hand operand of type 'class Person' (or there is no acceptable conversion)

- What would happen if the above is allowed?

The extra fields in the derived class would become uninitialized.



- Summary

"Derived class object to base class object" loses data (but is legal).
"Base class obj to derived class obj" leaves data uninitialized (illegal).

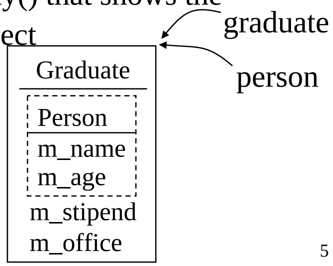
Assignment to Base Class Pointer

- Assignment from a derived pointer to a base class pointer is **legal**

```
Person      *person = new Person("Joe", 19);
Graduate  *graduate = new Graduate("Michael", 24, 6000, "8899 Storkes");
person–>display();
person = graduate;
person–>display();
```

Output
Joe is 19 years old.
Michael is 24 years old.

- What happened
  1. person–>display() calls Person::display() that shows the private data of the Base part of the object pointed to by the pointer ***graduate***
  2. Person::display() cannot access Graduate::m_stipend and Graduate::m_office



Assignment to Derived Class Pointer

- Assignment from a base pointer to a derived pointer is **illegal**, but you certainly can coerce it with an explicit type cast

```
Person          *person = new Person("Joe", 19);
Graduate       *grad1, *grad2=new Graduate("Michael", 24, 6000, "8899 Storkes");
grad1 = (Graduate *) person;
grad1–>display();
```

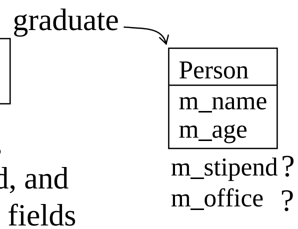
Output
Joe is 19 years old.
He is a graduate student.
He has a stipend of –384584985 dollars.
His address is 324rekj8

- This is called a downcast. Downcast is dangerous. It is correct only when the object pointed by *person* is an object of class Graduate.

```
ex. person = grad2;
       …
    grad1 = (Graduate *) person;
```

- What happened:

  grad1–>display() calls Graduate::display(), which accesses m_name, m_age, m_stipend, and m_office to display them, but the latter two fields do not exist for this Person object



Heterogeneous Container

- We would like to store all types of objects in a single database/array.

```
Person *database[3];
database[0] = new Undergraduate("Bob", 18);
database[1] = new Graduate("Michael", 25, 6000, "8899 Storkes");
database[2] = new Faculty("Ron", 34, "Gates 199", "associate professor");
for (int i=0; i<3; i++)
   database[i]->display();
```

Output
Bob is 18 years old.
Michael is 25 years old.
Ron is 34 years old.

- What is called by the above code is always Person::display() which shows only the Base part of each object instead of the display() member function of the derived class which shows all detail information of the derived class.

  Note: in the above program, we can use static object array Person database[3]; as well, the printed result would be the same, but what are really saved differ.

- Is there a modification that can make the above code display all detail information of any derived class in a uniform way?

A Solution with Data Tag

- Create an enumerated type for each base type:
  enum ObjectType {undergrad, grad, professor};
- Add a tag of this type to the base class

```
class Person {
public:
   Person();
   Person(char *name, int age, ObjectType typeTag);
   ~Person();
   ObjectType getType();
   void display() const;
private:
   char *m_name;
   int m_age;
   ObjectType m_typeTag;
};
```

```
Undergraduate::Undergraduate(…):
   Person(…,undergrad)
{…}
```

- Make the necessary changes in the constructor

```
Person::Person(char *name, int age, ObjectType typeTag)
         : m_age(age), m_typeTag(typeTag) {
   m_name = new char[strlen(name)+1];
   strcpy(m_name, name);
}
```

A Solution with Data Tag (Cont'd)

```
Person *database[3], *temp;
database[0] = new Undergraduate("Bob", 18);
database[1] = new Graduate("Michael", 25, 6000, "8899 Storkes");
database[2] = new Faculty("Ron", 34, "Gates 199", "associate professor");
for (int i=0; i<3; i++)
{
   temp = database[i];
   switch (temp->getType())
   {
   case undergrad:
      ((Undergraduate *) temp)->display(); // this is downcast
      break;
   case grad:
      ((Graduate *) temp)->display(); // this is downcast
      break;
   case professor:
      ((Faculty *) temp)->display(); // this is downcast
      break;
   }
}
```

Using code to select code

Downcast is the "goto" for OOP!!

Solution with Virtual Function

- Declare the function as *virtual* in the base class

```
class Person {
public:
   Person();
   Person(char *name, int age);
   ~Person();
   virtual void display() const;
private:
   char *m_name;
   int m_age;
};
```

Output
Bob is 18 years old.
He is an undergraduate.
Michael is 25 years old.
He is a graduate student.
He has a stipend of 6000 dollars.
His address is 8899 Storkes.
Ron is 34 years old.
His address is Gates 199.
His rank is associate professor.

- The rest of the code is all the same

```
Person *database[3];
database[0] = new Undergraduate("Bob", 18);
database[1] = new Graduate("Michael", 25, 6000, "8899 Storkes");
database[2] = new Faculty("Ron", 34, "Gates 199", "associate professor");
for (int i=0; i<3; i++)
   database[i]->display();
```

or equivalently

(*database[i]).display();

Will invoke Undergraduate::display() Graduate::display() and Faculty::display() in turn

Function Pointer

- Increasing the flexibility of your program
- Making the process / mechanism an adjustable parameter (you can pass a function pointer to a function) ex. qsort(), find(), sort()
- Syntax:

   return_type (*function_pointer_variable)(parameters);

- Example:

```
int func1(int x) {          int func2(int x) {
   …                            …
   return 0;                    return 0;
}                            }
int (*fp)(int);
fp = func1;
(*fp)(123); // calling function func1(), i.e. func1(123)
```


Function Pointer (cont'd)

- Increasing the **flexibility** of the program
- Example continued

func1(), func2(), and fp are defined as before

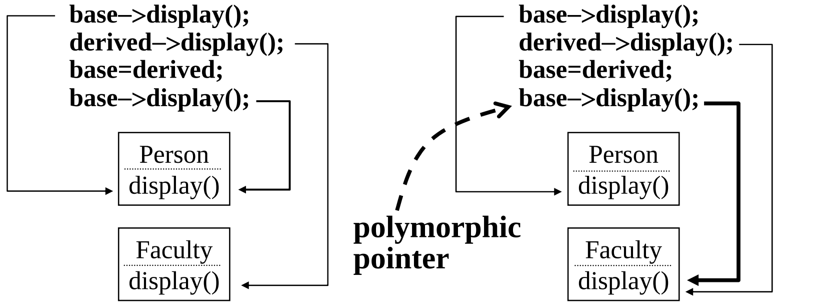
Consider the following function:

```
void service(int (*proc)(int), int data) {
   …
   (*proc)(data);
   …
}
…
fp = func2;
…
service(fp, x);
```

Virtual vs. Non-virtual Functions

Person *base = new Person("Bob", 18);
Faculty *derived = new Faculty("Ron", 34, "Gates 199", "associate professor");

Nonvirtual function | Virtual function



Compile-time binding

Run-time binding
(Late-binding, dynamic binding)

The function to be called is determined by the type of the pointer during compilation.

The function to be called is determined by the object the pointer refers to during run-time.

Virtual Function

✧ The keyword *virtual* is not required in any derived class.

```
class Undergraduate: public Person {
public:
   Undergraduate(char *name, int age);
   virtual void display() const;    // optional here if display() is already a virtual
};                                  //  function in Person class
```

Some C++ programmers consider it a good style to include the keyword for clarity

✧ Syntax
The keyword *virtual* must not be used in the function definition, only in the declaration

error C2723: 'func1' : 'virtual' storage-class specifier illegal on function definition

✧ Historical backgrounds

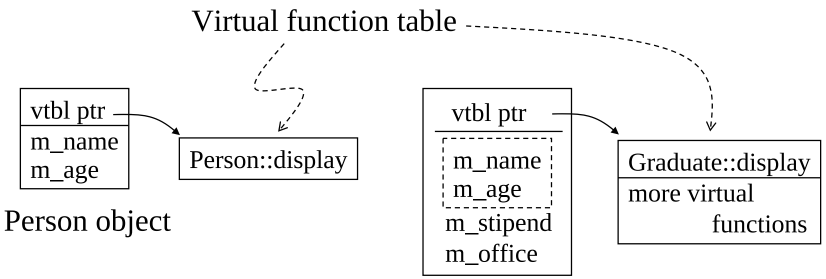
- Most object-oriented languages have only run-time binding.
- C++, because of its origins in C, has compile-time binding by default.

Efficiency consideration

✧ **Static member functions** and **constructors** cannot be declared virtual.   **Destructors** are always declared as virtual functions.

Virtual Function Table

✧ C++ use function pointers to implement the late binding (runtime binding) mechanism of virtual functions: the address of virtual member functions are stored in each object as a data structure “virtual function table” as follows



a Graduate object

Note: addresses of non-virtual functions are not kept in the virtual function table

Virtual Function vs. Overloading

✧ **Overloading** (**static** polymorphism or **compile-time** polymorphism)

```
void Person::display() const;
void Person::display(bool showDetail) const;
```

The arguments of the overloaded functions must differ.

✧ **Overriding** (virtual functions, **dynamic** polymorphism)

```
virtual void Person::display() const;
virtual void Faculty::display() const;
```

The arguments must be identical.

Note that scope operators are **not** required in these declarations, they are only for illustration purpose.

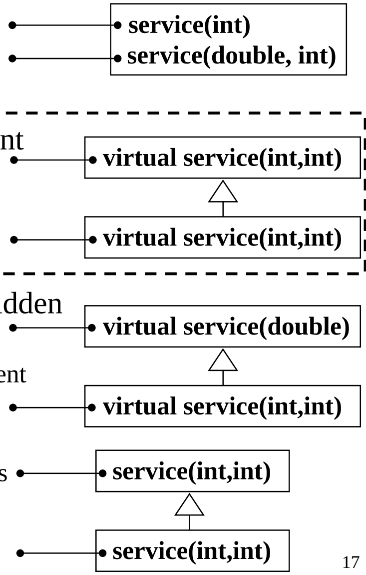
✧ What happens if the arguments are not identical?

```
virtual void Person::display() const;
virtual void Faculty::display(bool showDetail) const;
```

- In Faculty class, display(bool) does *not override* Person::display(),
- It does **NOT** *overload* Person::display() either.
- This phenomenon is called ***hiding***.
- Only Faculty::display(bool) exists in the Faculty class, there is no Faculty::display(), although Person::display() exists in its base class.

Overloading, Overriding, Hiding

- *Overloading*: two functions have the same scope, same name, different signatures (virtual is not required)
- *Overriding*: two functions have different scopes (parent vs child), same name, same signatures (virtual is required)
- *Hiding*: base class member function is hidden
  1. When a base class and a derived class declare virtual member functions with different signatures but with the same name.
  2. When a base class declares a non-virtual member function and a derived class declares a member function with the same signature.



Member Function Calling Mechanism

```
Faculty *prof = new Faculty("Ron", 34, "Gates 199", "associate professor");
Person *person = prof;
person–>display();      // dynamically binded, calling Person::display()
person–>display(true);// compile-time error, display() does not take 1 param
prof–>display();          // compile-time error, display(bool) does not take 0 param
prof–>display(true);    // dynamically binded, calling Faculty::display(bool)
```

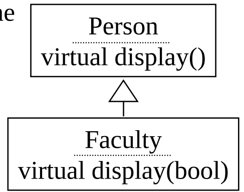
- The member function resolution and binding **rules** in C++:

  referrer.function()   referrer->function()

  1. Search in the scope of the static type of the referrer pointer/reference/object to find the specified function in its explicitly defined functions
  2. If it is a virtual function and referrer is a pointer (including ***this*** pointer) or reference, use dynamic binding otherwise use static one

What functions are explicit in the scope of a class?

1. **Defined in the class declaration**
2. **Search upward the inheritance tree, match all functions not hided previously (by any function having the same name)**



Explicitly Defined Functions

Virtual functions: 2, 4, 5, 6, 8, 9, 10, 11

```
class Base {
public:
   void func1() { cout << "Base::func1() #1\n"; }
   virtual void func2() { cout << "Base::func2() #2\n"; }
   void func3() { cout << "Base::func3() #3\n"; }
   virtual void func4() { cout << "Base::func4() #4\n"; }
   virtual void func5() { cout << "Base::func5() #5\n"; }
   virtual void func5(int, int) { cout << "Base::func5(int,int) #6\n"; }
};
```

Explicit: 1,2,3,4,5,6

```
class Derived: public Base {
public:
   void func3() {
      cout << "Derived::func3() #7\n";
   }
   void func4() {
      cout << "Derived::func4() #8\n";
   }
   void func5(int) {
      cout << "Derived::func5(int) #9\n";
   }
};
```

Explicit: 1,2,7,8,9
Implicit: 3,4,5,6

```
class FDerived1: public Derived {
};
```

Explicit: 1,2,7,8,9
Implicit: 3,4,5,6

```
class FDerived2: public Derived {
public:
   void func5() {
         cout << "FDerived2::func5() #10\n";
   }
   void func5(int, int) {
      cout << "FDerived2::func5(int, int) #11\n";
   }
};
```

Explicit: 1,2,7,8,10,11
Implicit: 3,4,5,6,9

Function Call Resolving (1/11)

```
class Base {
public:
   void func();
};

class Derived: public Base {
};

void main() {
   Derived d, *dp=&d;
   Base b, *bp1=&b, *bp2=&d;

   b.func();          // static binding
   bp1–>func();   // static binding

   bp2–>func();   // static binding
   d.func();          // static binding
   dp–>func();     // static binding
}
```

Function Call Resolving (2/11)

```
class Base {
public:
   virtual void func();
};

class Derived: public Base {
};

void main() {
   Derived d, *dp=&d;
   Base b, *bp1=&b, *bp2=&d;

   b.func();        // static binding
   bp1–>func();  // dynamic binding  Base::func()

   bp2–>func();  // dynamic binding  Base::func()
   d.func();        // static binding
   dp–>func();    // dynamic binding  Base::func()
}
```


Function Call Resolving (3/11)

```
class Base {
public:
   virtual void func();
};

class Derived: public Base {
public:
   virtual void func();
};

void main() {
   Derived d, *dp=&d;
   Base b, *bp1=&b, *bp2=&d;

   b.func();        // static binding
   bp1–>func();  // dynamic binding  Base::func()

   bp2–>func();  // dynamic binding  Derived::func()
   d.func();        // static binding
   dp–>func();    // dynamic binding  Derived::func()
}
```


Function Call Resolving (4/11)

```
class Base {
public:
   virtual void func();
};

class Derived: public Base {
private:
   virtual void func();
};

void main() {
   Derived d, *dp=&d;
   Base b, *bp1=&b, *bp2=&d;

   b.func();        // static binding
   bp1–>func();  // dynamic binding  Base::func()

   bp2–>func();  // dynamic binding  Derived::func() violate the access restriction
   //d.func();      // error in accessing private member
   //dp–>func();  // error in accessing private member
}
```


Function Call Resolving (5/11)

```
class Base {
public:
   virtual void func();
};

class Derived: public Base {
public:
   virtual void func(int);
};

void main() {
   Derived d, *dp=&d;
   Base b, *bp1=&b, *bp2=&d;

   b.func();        // static binding
   bp1–>func();  // dynamic binding  Base::func()

   bp2–>func();  // dynamic binding  Base::func()
   //d.func();      // error func() does not take zero param
   //dp–>func();  // error func() does not take zero param

   //b.func(1);     // error func() does not take one param
   //bp1–>func(1);// error func() does not take one param

   //bp2–>func(1);// error func() does not take one param
   d.func(1);       // static binding
   dp–>func(1);   // dynamic binding  Derived::func(int)
}
```

Function Call Resolving (6/11)

```
class Base {
public:
   virtual void func();
};

class Derived: public Base {
public:
   void func();
   virtual void func(int);
};

void main() {
   Derived d, *dp=&d;
   Base b, *bp1=&b, *bp2=&d;

   b.func();         // static binding
   bp1–>func();  // dynamic binding  Base::func()

   bp2–>func();  // dynamic binding  Derived::func()
   d.func();         // static binding, Derived::func()
   dp–>func();    // dynamic binding, Derived::func()

   //b.func(1);      // error func() does not take one param
   //bp1–>func(1);// error func() does not take one param

   //bp2–>func(1);// error func() does not take one param
   d.func(1);       // static binding
   dp–>func(1);  // dynamic binding  Derived::func(int)
}
```


Function Call Resolving (7/11)

```
class Base {
public:
   virtual void func();
   virtual void func(int);
};

class Derived: public Base {
};

void main() {
   Derived d, *dp=&d;
   Base b, *bp1=&b, *bp2=&d;

   b.func();         // static binding
   bp1–>func();  // dynamic binding  Base::func()

   bp2–>func();  // dynamic binding  Base::func()
   d.func();         // static binding, Base::func()
   dp–>func();    // dynamic binding, Base::func()

   b.func(1);       // static binding, Base::func(int)
   bp1–>func(1);// dynamic binding, Base::func(int)

   bp2–>func(1);// dynamic binding Base::func(int)
   d.func(1);       // static binding Base::func(int)
   dp–>func(1);  // dynamic binding  Base::func(int)
}
```


Function Call Resolving (8/11)

```
void main() {
   Derived d, *dp=&d;
   Base b, *bp1=&b, *bp2=&d;

   b.func();            // static binding
   bp1–>func();     // dynamic binding  Base::func()

   bp2–>func();     // dynamic binding  Base::func()
   //d.func();          // error func() does not take 0 param
   //dp–>func();     // error func() does not take 0 param

   b.func(1);          // static binding, Base::func(int)
   bp1–>func(1);   // dynamic binding, Base::func(int)

   bp2–>func(1);   // dynamic binding Base::func(int)
   //d.func(1);        // error func() does not take 1 param
   //dp–>func(1);   // error func() does not take 1 param

   //b.func(1, 1);     // error func() no overloaded function take 2 param
   //bp1–>func(1, 1);// error func() no overloaded function take 2 param

   //bp2–>func(1, 1);// error func() no overloaded function take 2 param
   d.func(1, 1);       // static binding, Derived::func(int, int)
   dp–>func(1, 1);  // dynamic binding, Derived::func(int, int)
}

class Base {
public:
   virtual void func();
   virtual void func(int);
};

class Derived: public Base {
public:
   virtual void func(int, int);
};
```


Function Call Resolving (9/11)

```
void main() {
   Derived d, *dp=&d;
   Base b, *bp1=&b, *bp2=&d;

   b.func();            // static binding Base::func()
   bp1–>func();     // dynamic binding  Base::func()

   bp2–>func();     // dynamic binding  Derived::func()
   d.func();            // static binding Derived::func()
   dp–>func();       // dynamic binding  Derived::func()

   b.func(1);          // static binding, Base::func(int)
   bp1–>func(1);   // dynamic binding, Base::func(int)

   bp2–>func(1);   // dynamic binding Base::func(int)
   //d.func(1);        // error func() does not take 1 param
   //dp–>func(1);   // error func() does not take 1 param

   //b.func(1, 1);     // error func() no overloaded function take 2 param
   //bp1–>func(1, 1);// error func() no overloaded function take 2 param

   //bp2–>func(1, 1);// error func() no overloaded function take 2 param
   d.func(1, 1);       // static binding, Derived::func(int, int)
   dp–>func(1, 1);  // dynamic binding, Derived::func(int, int)
}

class Base {
public:
   virtual void func();
   virtual void func(int);
};

class Derived: public Base {
public:
   virtual void func();
   virtual void func(int, int);
};
```

Function Call Resolving (10/11)

```
void main() {
   FurtherDerived fd, *fdp=&fd;
   Derived d, *dp1=&d, *dp2=&fd;
   Base b, *bp1=&b, *bp2=&d, *bp3=&fd;
   b.func();          // static binding   Base::func()
   bp1->func();       // dynamic binding  Base::func()
   //d.func();        // error func() does not take zero param
   //dp1->func();     // error func() does not take zero param
   //dp2->func();     // error func() does not take zero param
   bp2->func();       // dynamic binding  Base::func()
   fd.func();         // static binding  FurtherDerived::func()
   fdp->func();       // dynamic binding  FurtherDerived::func()
   bp3->func();       // dynamic binding  FurtherDerived::func()
   b.func(1);         // static binding   Base::func(int)
   bp1->func(1);      // dynamic binding  Base::func(int)
   //d.func(1);       // error func() does not take 1 param
   //dp1->func(1);    // error func() does not take 1 param
   //dp2->func(1);    // error func() does not take 1 param
   bp2->func(1);      // dynamic binding  Base::func()
   //fd.func(1);      // error func() does not take 1 param
   //fdp->func(1);    // error func() does not take 1 param
   bp3->func(1);      // dynamic binding  Base::func(int)
```

```
class Base {
public:
   virtual void func();
   virtual void func(int);
};

class Derived: public Base {
public:
   virtual void func(int, int);
};
class FurtherDerived:
               public Derived {
public:
   virtual void func();
};
```


Function Call Resolving (11/11)

```
class Base {
public:
   virtual void func();
   virtual void func(int);
};
```

```
class Derived: public Base {
public:
   virtual void func(int, int);
};
```

```
class FurtherDerived:
               public Derived {
public:
   virtual void func();
};
```

```
   //b.func(1, 2);         // error func() does not take 2 param
   //bp1->func(1, 2);      // error func() does not take 2 param

   d.func(1, 2);           // static binding  Derived::func(int, int)
   dp1->func(1, 2);        // dynamic binding  Derived::func(int, int)
   dp2->func(1, 2);        // dynamic binding  Derived::func(int, int)
   //bp2->func(1, 2);      // error func() does not take 2 param

   //fd.func(1, 2);        // error func() does not take 2 param
   //fdp->func(1, 2);      // error func() does not take 2 param
   //bp3->func(1, 2);      // error func() does not take 2 param
}
```


Polymorphism

- **Polymorphism**: a single identity stands for different things
- C++ implements polymorphism in three ways
  - **Overloading** – ad hoc polymorphism (static polymorphism)
    one name stands for several functions
  - **Templates** – parameterized polymorphism
    one name stands for several types or functions
  - **Virtual functions** – pure polymorphism (dynamic polymorphism)
    one pointer refers to any base or derived class objects
    use object to select code
- Many OO languages does not support parameterized polymorphism, e.g. JAVA before J2SE 5.0 (2004), it is called *Generics* in Java
- Is there any drawback to pure polymorphism?
  Virtual function calls are less efficient than non-virtual functions
- What are the benefits from polymorphism?
  Superior abstraction of object usage (code reuse), old codes call new codes (usage prediction)

Code Reuse

- There are basically two major types of code reuse:
  - Library subroutine calls: put all repeated procedures into a function and call it whenever necessary. The codes gathered into the function is to be reused.
    Note: basic inheritance syntax would automatically include all data members and member functions of parent classes into the child class. This is also a similar type of program reuse.
  - Factoring: sometimes, we substitute a particular module in a program with a replacement. In this case, the other part of system is reused.
    Note: ex. 1. OS patches or device drivers replace the old module and reuse the overall architecture.
    2. Application frameworks provide the overall application architectures while programmer supply minor modifications and features.
    interface inheritance also reuses the other part of program.

Old Codes Call New Codes

- Using existent old codes to call non-existent new codes
- Using data (object) to select codes
- While writing the following codes, the programmer might not know which display() function is to be called. The actual code be called might not existent at the point of writing. He only knows that the object pointed by database[i] must be inherited from Person. The semantics of the virtual function display() is largely determined in designing the class Person. The derived class should not change it.

```
void show(Person *database[3]) {
   for (int i=0; i<3; i++)
      database[i]->display();
}
```

old (current) codes

Later, if we derive a class Staff from Person, and implement a new member function Staff::display(), ← new codes

```
database[0] = new Staff(…);
…
show(database);
```


Two Major Code Reuses of Inheritance

- Code inheritance: reuse the data and codes in the base class
- Interface inheritance: reuse the codes that employ(operate) the base class objects
- Comparing the above two types of code reuse, the first one reuses only considerable amount of old codes. The second one usually reuses a bulk amount of old codes.
- Interface inheritance is a very important and effective way of reusing existent codes. This feature makes Object Oriented programming successful in the framework design, in which the framework provides a common software platform, ex. Window GUI environment, math environment, or scientific simulation environment. Using predefined interfaces (abstract classes in C++), a framework can support all utility functions to an empty application project.

Using C++ Polymorphism

- Should you make every (non-private) function virtual?
  - Some C++ programmers do.
  - Others do so only when compelled by necessity.
  - Java's member function are all virtual.
  - Doing so ensures the pure OO semantics and have good semantic compatibility if you are using multiple OO languages.
  - You can change to non-virtual when profiling shows that the overhead is on the virtual function calls

Virtual Function vs. Inline Function

- Virtual function and inline function are contradicting language features
  - Virtual function requires runtime binding but inline function requires compile-time code expansion
- However, you will see in many places virtual inline combinations, ex.

```
class base {
  …
  virtual ~base() {}
  …
};
```

- Why??

  Virtual function does not always use dynamic binding.
  This is a C++ specific feature.

Virtual Function vs. Static Function

- Virtual function and static function are also contradicting language features
  - Static function is a class method shared among all objects of the same class. Calling a static function does NOT mean sending a message to an object. There is no “this” object in making a static function call.
  - It is, therefore, completely useless to put a static function in the virtual function table. (calling a static function does not require a target object, and thus the virtual function table within it)
  - A static function cannot be virtual. Calling a static function always uses static binding. No overriding with static function.
  - You can redefine a static function in a derived class. The static function in the base class is *hided* as usual.

Virtual Destructors

- Base classes and derived classes may each have destructors

```
Person::~Person() {
   delete[] m_name;
}
Faculty::~Faculty() {
   delete[] m_rank;
}
```

- What happens in this scenario?

```
Person *database[3];
Faculty *prof = new Faculty("Ron", 40, "6000 Holister", "professor");
database[0] = prof;
delete database[0];
```

  - If the destructor of Person is non-virtual, only the destructor for Person will be called, the Faculty part of the object will not be destructed suitably.
- The solution is simple

```
virtual ~Person();    // virtual destructor
```

  - Note: This syntax makes every destructor of every derived class virtual even though the names do not match. Visual Studio automatically does this.