Polymorphism

C++ Object Oriented Programming

Pei-yih Ting

NTOU CS

Contents

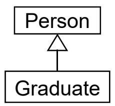
Assignment to Base Class Object

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

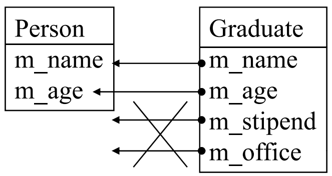
```
Person      person("Joe", 19);
Graduate graduate("Michael", 24, 6000, "INS501");
person.display();
person = graduate; // assign
person.display();
Person  person2 = graduate; // copy
person2.display();
```

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**

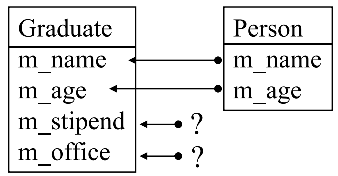
~~**graduate = person; // assignment**~~

~~**Graduate graduate2 = person; // copy ctor**~~

```
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** to **base**" only loses data (allowed).

  "**base** to **derived**" leaves state undefined (forbidden).

Assignment to Base Class Pointer

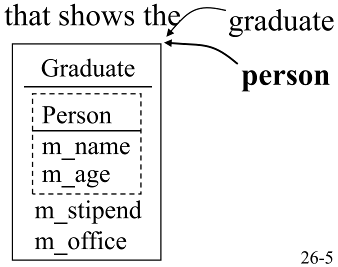
- Assignment from a **derived** pointer to a **base** class pointer is **legal**

```
Person     *person = new Person("Joe", 19);
Graduate  *graduate = new Graduate("Michael", 24, 6000, "INS501");
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 either ***Person*** or ***Graduate*** object
  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 **evil**, 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, "INS501");
grad1 = (Graduate *) person;
grad1–>display();
```

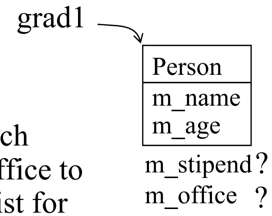
Output

```
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 a Person object **grad1=dynamic_cast<Graduate *> person; //grad1 will be 0**



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, "INS501");
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**() of a 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 results would be the same, but what it really saved differ.

- Is there a way that can make the above code display all detail information of every 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, "INS501");
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();
    break;
  case grad:
    ((Graduate *) temp)->display();
    break;
  case professor:
    ((Faculty *) temp)->display();
    break;
  }
}
```

evil

Downcast is the "goto" for OOP!!

Using code to select code

another way to implement w/o tags

```
// RTTI
if (dynamic_cast<Undergraduate*>(temp))
  ((Undergraduate*)temp)->display();
else if (dynamic_cast<Graduate*>(temp))
  ((Graduate*)temp)->display();
else if (dynamic_cast<Faculty*>(temp))
  ((Faculty*)temp)->display();
```

This is a segment of code not satisfying **open-closed principle**. Usually, this is avoided with the "**strategy**" pattern.

Solution with Virtual Function

① 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;
};
```

Outputs

Bob is 18 years old.
He is an undergraduate.

Bob is 18 years old.
He is a graduate student.
He has a stipend of 6000 dollars.
His address is INS501.

Ron is 34 years old.
His address is INS512.
His rank is associate professor.

② **Invoke indirectly through <u>base class pointer</u> or <u>reference</u>**

```
Person *database[3];
database[0] = new Undergraduate("Bob", 18);
database[1] = new Graduate("Michael", 25, 6000, "INS501");
database[2] = new Faculty("Ron", 34, "INS512", "associate professor");
for (int i=0; i<3; i++)
   database[i]->display();
```

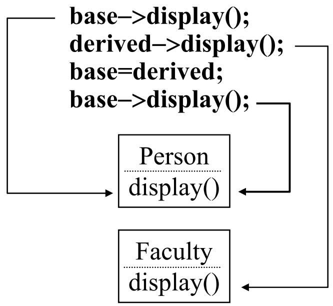
Will invoke Undergraduate::display(), Graduate::display(), and Faculty::display() in turn

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

Virtual vs. Non-virtual Functions

```
Person *base = new Person("Bob", 18);
Faculty *derived = new Faculty("Ron", 34, "INS512", "associate professor");
```

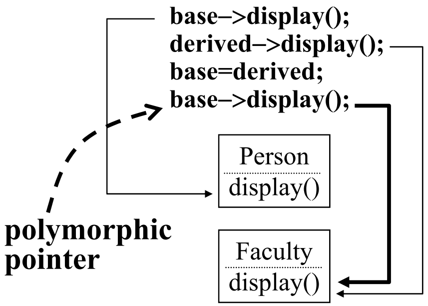
Nonvirtual function



Compile-time binding (static binding)

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

Virtual function



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

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 a 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. (e.g. Java)
  - C++, because of its origin 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.

Function Pointer

- Increasing the **flexibility of your program** making the **algorithm** / **mechanism** an adjustable parameter (you can pass a function pointer to a function) ex. qsort(), find(), sort()
- **Higher order programming**: functions are objects that can be input/output of some algorithms and processed extensively.
- **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)

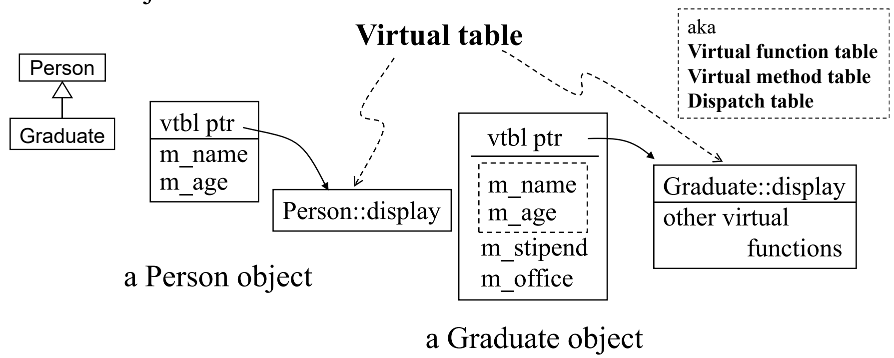
- Increasing the **flexibility** of the program
- Example continued

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

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


Virtual Table

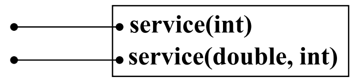
- C++ uses function pointers to implement the **late binding** (**runtime binding**, **dynamic binding**, **dynamic dispatch**) mechanism of virtual functions: the address of virtual member functions are stored in each object as the "virtual table" data structure



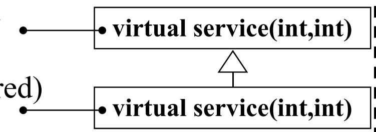
Note: addresses of non-virtual functions are not kept in the virtual table

Overloading, Overriding, Hiding

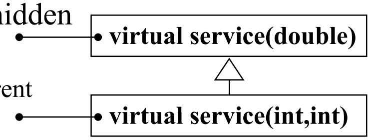
- ***Overloading***: two functions in the same scope, have the same name, different signatures (virtual is not required)



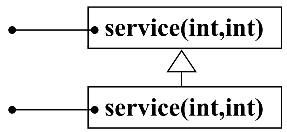
- ***Overriding***: two functions in different scopes (parent vs child), have the same name. same signatures (**virtual** is required)



- ***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 name but **with** or **without** the same signature.



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;
  ```

  Person::display()
  ~~Faculty::display()~~
  Faculty::display(bool)

  - In Faculty class, display(bool) does **NOT** *override* Person::display(),
  - It does **NOT** *overload* Person::display() neither.
  - This phenomenon is called ***hiding***.
  - Only **Faculty::display(bool)** can be called with a Faculty object or pointer, no Faculty::display(), although **Person::display()** can also be called.

Member Function Calling Rule

```
Faculty *prof = new Faculty("Ron", 34, "INS512", "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 hidden/overridden previously (by any function having the same name)**

Person
virtual display()

Faculty
virtual display(bool)

Explicitly Defined Functions

```
class Base {
public:
   void funcA() { cout << "Base::funcA() #1\n"; }
   virtual void funcB() { cout << "Base::funcB() #2\n"; }
   void funcC() { cout << "Base::funcC() #3\n"; }
   virtual void funcD() { cout << "Base::funcD() #4\n"; }
   virtual void funcE() { cout << "Base::funcE() #5\n"; }
   virtual void funcE(int, int) { cout << "Base::funcE(int,int) #6\n"; }
};
```

Virtual table: 2, 4, 5, 6

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

```
class Derived: public Base {
public:
   void funcC() {
      cout << "Derived::funcC() #7\n";
   }
   void funcD() {
      cout << "Derived::funcD() #8\n";
   }
   virtual void funcE(int) {
      cout << "Derived::funcE(int) #9\n";
   }
};
```

Virtual table: 2, 8, 5, 6, 9

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

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

Virtual table: 2, 8, 5, 6, 9

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

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

Virtual table: 2, 8, 10, 11, 9

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

Polymorphism

- **Polymorphism**: a single identifier stands for different things
- C++ implements polymorphism in three ways
  - **Overloading** – ad hoc / static polymorphism, static dispatch
    one name stands for several functions
  - **Templates** – parametric polymorphism
    one name stands for several types or functions
  - **Virtual functions** – pure / dynamic polymorphism, dynamic dispatch
    one pointer (reference) 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 Paradigms

- There are basically two major types of code reuses:
  - **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

- Make existent **old codes** 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 exist 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 codes
closed for modification
but **open** for extension

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

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

new codes

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
- The first one reuses only considerable amount of old codes. The second one usually reuses a vast 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 designing a framework, which provides a common software development platform, ex. Window GUI environment, math environment, video processing, 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 it is necessary.
  - 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 (compile-time binding makes the code run faster) when profiling shows that the overhead comes from 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 hidden "this" pointer 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 *hidden* 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, "INS312", "professor");
database[0] = prof;
delete database[0];
```

  - If the destructor of Person is non-virtual, only the destructor of 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.

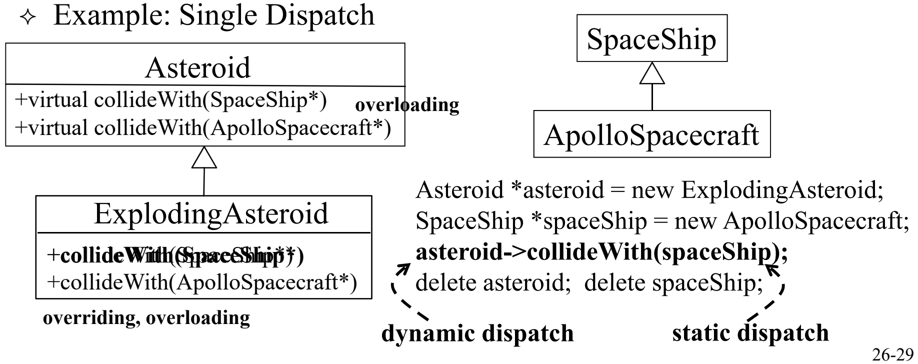
Invoking a Virtual function in Ctor/Dtor

- When invoking any virtual function of this class from inside a constructor or destructor (explicitly through *this* pointer or implicitly), **compile-time binding** will determine which method is called as if the *virtual* keyword was not there.
- In the process of constructing an object, the virtual table may be incompletely set up by the constructor especially for a derived object. Also, the virtual table may be partially or completely destroyed by the destructor in the process of destructing an object instance.
- It is not reasonable to expect runtime binding to work properly under these conditions.

Single / Double Dispatch

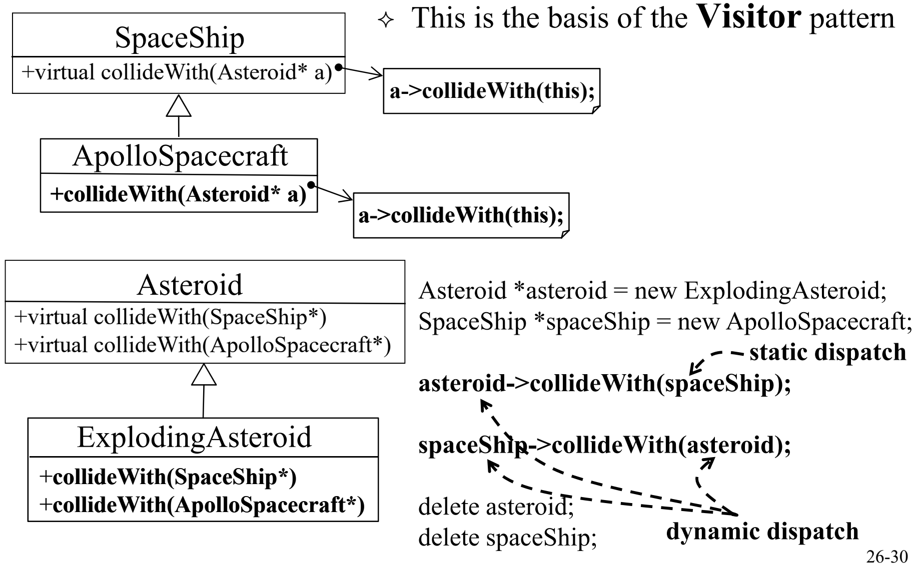
x->message(y);

- C++ (virtual) function provides only **single dispatch**: the decision of which **message()** to call is based on **the type of x (*x)**
- **Double dispatch**: the decision is based not only on **the type of x** but also on **the type of y**, C++ does not support double dispatch
- Example: Single Dispatch



Double Dispatch (cont'd)

- This is the basis of the **Visitor** pattern



Visitor Pattern

- A way of separating an **algorithm** from an **object structure on which it operates** such that it is possible to **add new operations to existing object structures** without modifying those structures and enforcing the OCP.