# Lecture 14

# Object-Oriented Programming X

## Virtual Functions

Prof. Hyeong-Seok Ko
Seoul National University
Graphics & Media Lab

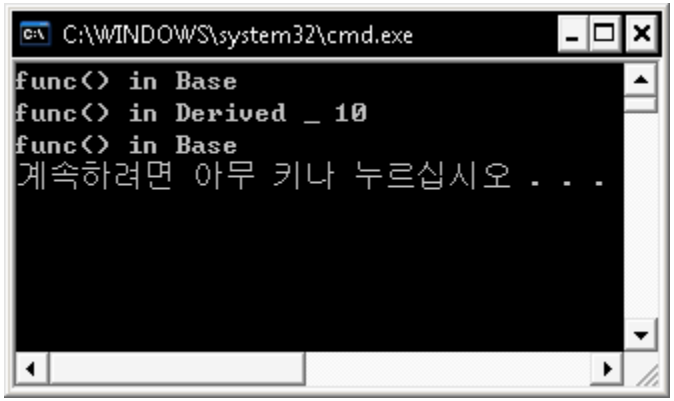
# Contents

- Name collisions (15.5.1~15.5.3)
- Pointer and Reference in Inheritance (15.5.4)
- Dynamic Binding & Virtual functions (15.2.4, 15.5.4)

# Name Collisions

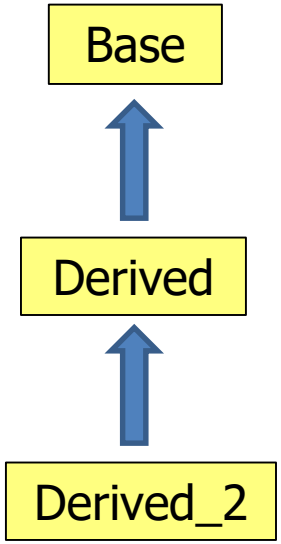
- A derived class member with the same name as a member of the base class hides direct access to the base-class member.
  - This hiding happens even when the member functions have different parameters.

```
class Base {
public :
   void func() {
      std::cout << "func() in Base" << std::endl;
   }
};

class Derived : public Base {
public :
   void func(int k) {    // hides func() in class Base
      std::cout << "func() in Derived _ " << k << std::endl;
   }
};

void main() {
   Base base; Derived derived;

   base.func();
   derived.func(10);

   derived.func();          // Compilation Error !!!
                            // func() does not take 0 argument.
                            // because Base::func() is hided by Derived::func()

   derived.Base::func();    // It is OK.

}
```



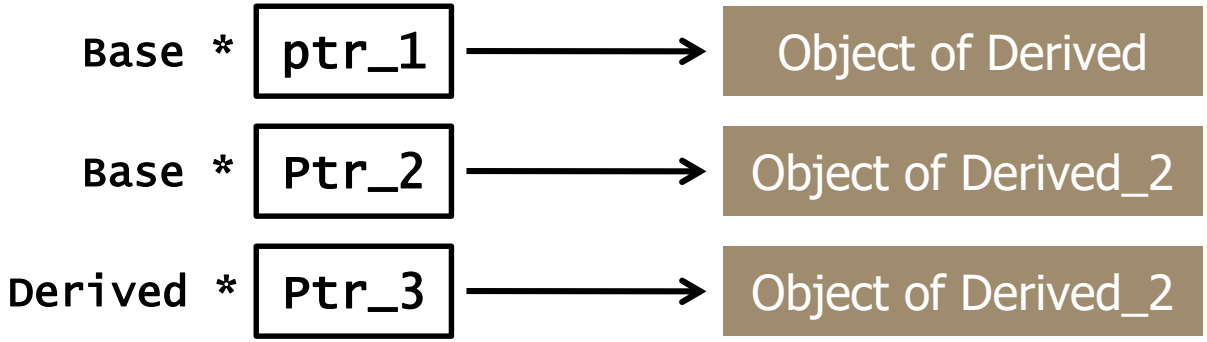
# Pointers in Inheritance

```
class Base {};

class Derived : public Base {};

class Derived_2 : public Derived {};

void main() {
   Base * ptr_1 = new Base();
   Derived * ptr_2 = new Derived();
   Derived_2 * ptr_3 = new Derived_2();
}
```

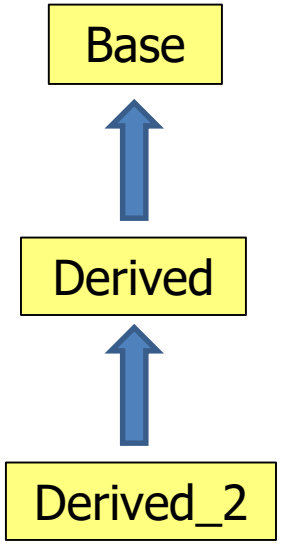


# Pointers in Inheritance

- The pointer (and also reference) of the **base class** can point the object of a **derived class**.

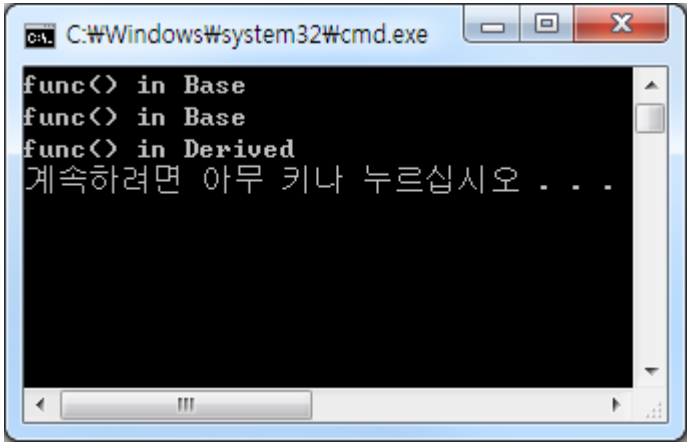


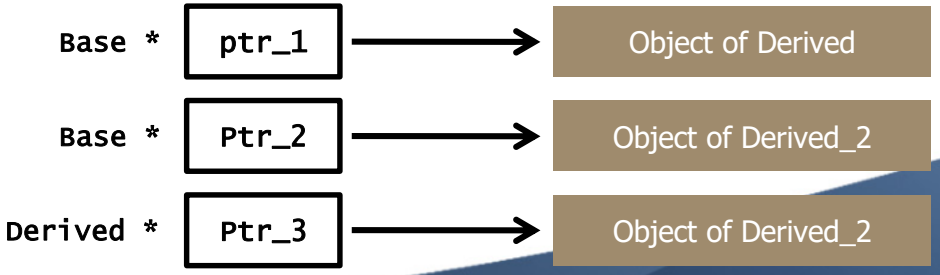

```
class Base {};

class Derived : public Base {};

class Derived_2 : public Derived {};

void main() {
   Base * ptr_1 = new Derived();        // It is OK.
   Base * ptr_2 = new Derived_2();      // It is OK.
   Derived * ptr_3 = new Derived_2();   // It is OK.
}
```



# Non-Virtual Function

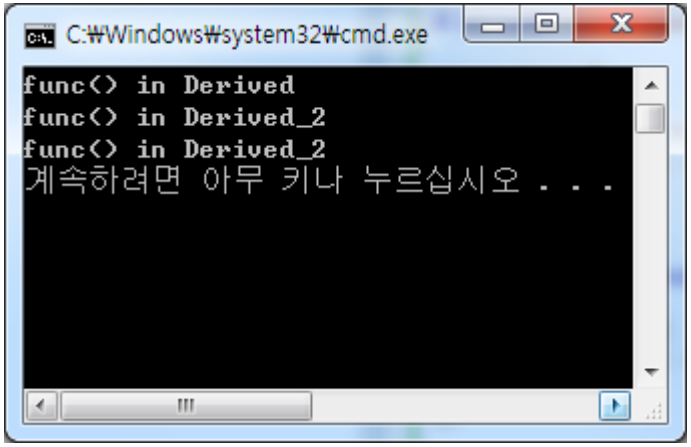
```
class Base {
public :
   void func() {
      std::cout << "func() in Base" << std::endl;
   }
};

class Derived : public Base {
public :
   void func() {
      std::cout << "func() in Derived" << std::endl;
   }
};

class Derived_2 : public Derived {
public :
   void func() {
      std::cout << "func() in Derived_2" << std::endl;
   }
};

void main() {
   Base * ptr_1 = new Derived();
   Base * ptr_2 = new Derived_2();
   Derived * ptr_3 = new Derived_2();
   ptr_1->func();
   ptr_2->func();
   ptr_3->func();
}
```

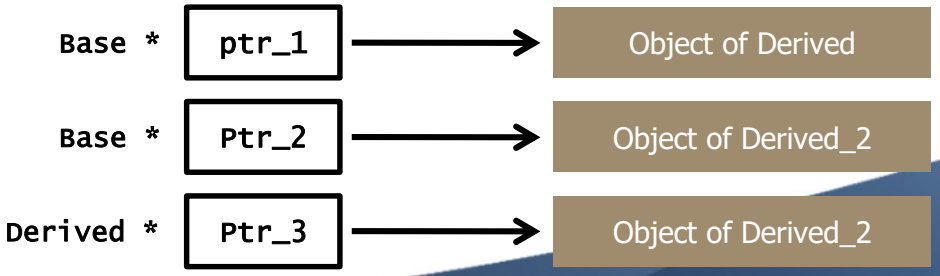


# Virtual Function

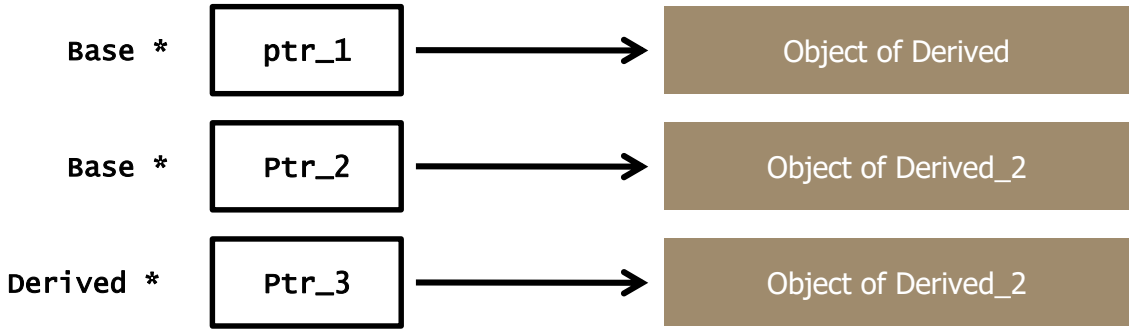
```
class Base {
public :
   virtual void func() {
      std::cout << "func() in Base" << std::endl;
   }
};

class Derived : public Base {
public :
   void func() {
      std::cout << "func() in Derived" << std::endl;
   }
};

class Derived_2 : public Derived {
public :
   void func() {
      std::cout << "func() in Derived_2" << std::endl;
   }
};

void main() {
   Base * ptr_1 = new Derived();
   Base * ptr_2 = new Derived_2();
   Derived * ptr_3 = new Derived_2();
   ptr_1->func();
   ptr_2->func();
   ptr_3->func();
}
```

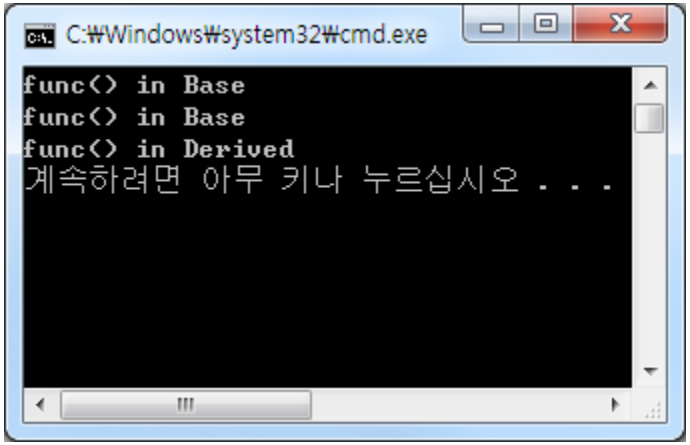


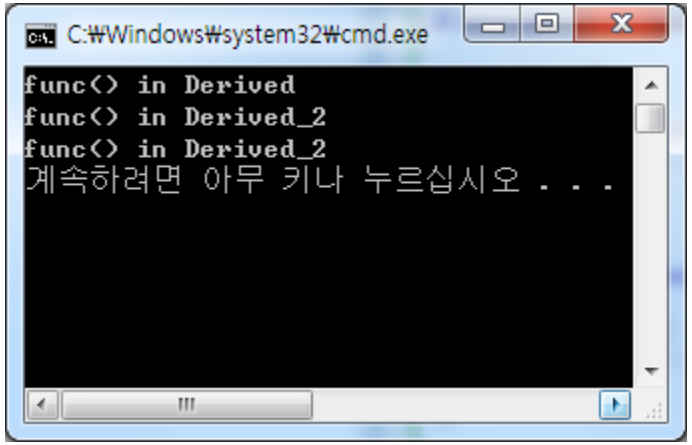
# Non-Virtual (vs) Virtual Function

- Comparison between **virtual** and **non-virtual** function.




**Non-virtual Function**




**Virtual Function**

# Dynamic Binding

- A call to a virtual function is handled according to the type of the actually allocated object rather than the type of the pointer. The above is called **dynamic binding** or **run-time polymorphism**.
  - In C++, you can call a **virtual** member function through a **pointer** of a base class.

# Brief overview of object-oriented programming

- Object-oriented programming (OOP) is a programming paradigm that uses **"objects"** – data structures consisting of **datafields** and **interfaces** together with their interactions – to design applications.
  - in Wikipedia _ http://en.wikipedia.org/wiki/Object_oriented

- The key features of OOP
  - Encapsulation
    - Data protection
    - Classes provide stable interfaces which protect the remainder of the program from the implementation (i.e., hides implementation details)
  - Inheritance
    - Hierarchy in the real world is reflected to the code.
    - Results in less amount of coding.
  - Dynamic Binding (Virtual Function)
    - Run-time polymorphism
    - A function call is handled according to the type of the actually allocated object rather than the type of the pointer.