DigiPen
INSTITUTE OF TECHNOLOGY

CS170#11.2

Multiple Inheritance

Vadim Surov

Copyright © 2017 DigiPen Corporation

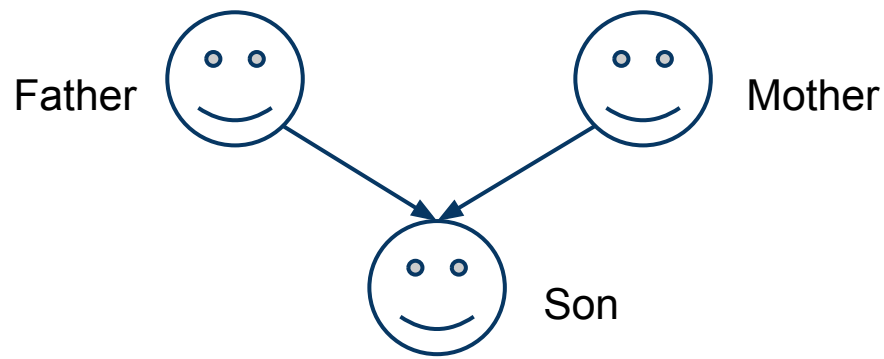
Outline

- Multiple Inheritance
  - Definition
  - Construction & Destruction
  - Conversions
  - Access To Members
  - Class Scope
  - Copy Control

Copyright © 2017 DigiPen Corporation

Multiple Inheritance

- Multiple inheritance is the ability to derive a class from more than one immediate base class



- A multiply derived class inherits the properties of all its ancestors
- Can present tricky design-level and implementation-level problems

Copyright © 2017 DigiPen Corporation

Defining Multiple Classes

```
class ZooAnimal { };
class Endangered { };
class Bear :
    protected ZooAnimal
{ };
class Panda :
    public Bear,
    Endangered
{ };
```

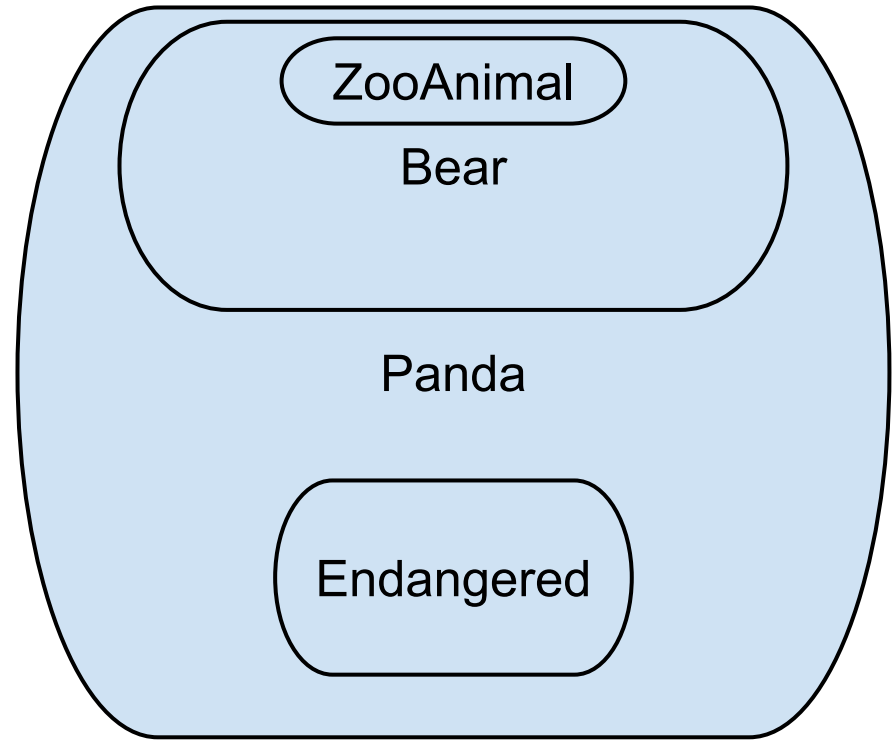
- No language-imposed limit on the number of base classes
- Base class may appear only once in a given derivation list
- Default access level is **private**

Copyright © 2017 DigiPen Corporation

Multiply Derived Classes Inherit State From Each Base Class

- Under multiple inheritance, objects of a derived class contain a base-class subobject for each of its base classes

- An object `Panda panda("Mimi");` is composed of a `Bear` class subobject, an `Endangered` class subobject, and data members, if any, declared within the `Panda` class



Copyright © 2017 DigiPen Corporation

Derived Constructors Initialize All Base Classes

- Constructing an object of derived type involves constructing and initializing all its base subobjects

```
//Explicitly initialize both base classes
Panda::Panda(std::string name, bool onExhibit)
    : Bear(name, onExhibit, "Panda"),
      Endangered(Endangered::critical) { }

//Implicitly use Bear default constructor to
initialize the bear subobject
Panda::Panda()
    : Endangered(Endangered::critical) { }
```

Copyright © 2017 DigiPen Corporation

Order Of Construction

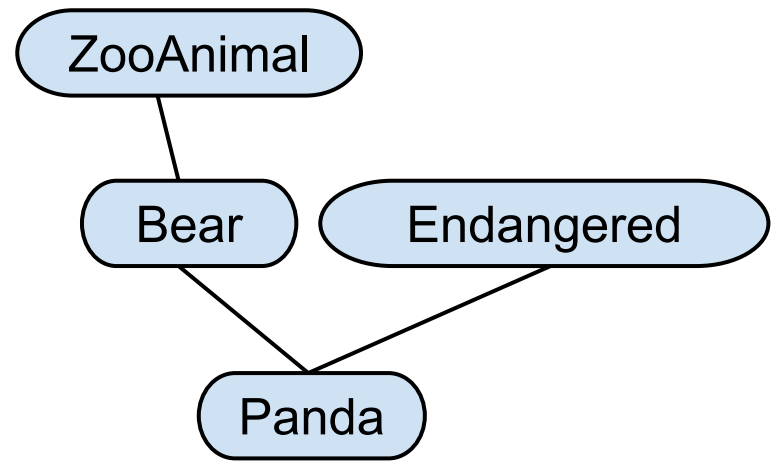
- The constructor **initialization list controls only the values** that are used to initialize the base classes
- The initialization list **do not control** the order in which the base classes are constructed
- **The base-class constructors are invoked in the order in which they appear in the class derivation list**

Copyright © 2017 DigiPen Corporation

Order Of Construction (contd)

- For `Panda`, the order of base-class initialization is:
  - `ZooAnimal`, the ultimate base class up the hierarchy from `Panda`'s immediate base class `Bear`
  - `Bear`, the first immediate base class
  - `Endangered`, the second immediate base, which itself has no base class
  - `Panda`, the members of `Panda` itself are initialized, and then the body of its constructor is run

```
class Endangered {};
class Bear :
  protected ZooAnimal
{};
class Panda :
   public Bear,
   Endangered
{};
```

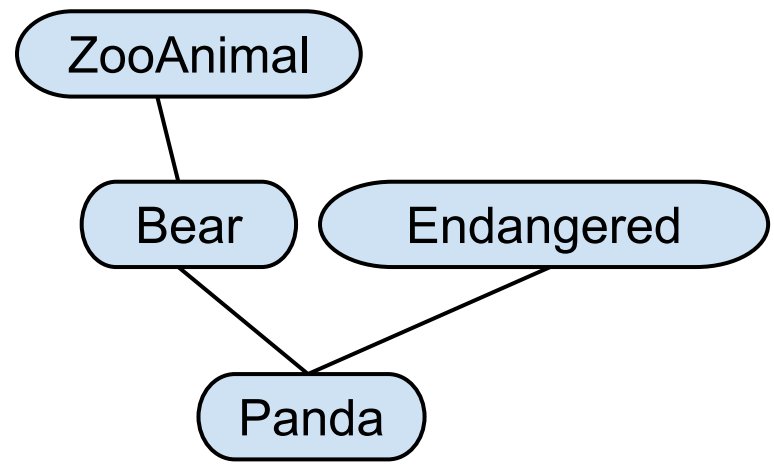


Copyright © 2017 DigiPen Corporation

Destruction

- The memberwise destruction of a multiply derived class behave in the same way as under single inheritance
- Base class destructors are always invoked **in the reverse order** from which the constructors are run
- In our example, the order in which the destructors are called is

```
~Panda(),
~Endangered(),
~Bear(),
~ZooAnimal()
```



Copyright © 2017 DigiPen Corporation

Virtual Destructor

- Witch destructor (Base or Derived or both) will be called?

```
Base     *basePtr     =     new     Derive();
delete basePtr;
```

- If base class defines their destructor as `virtual`, then derived class destructor will be called regardless of the pointer type through which we delete the object

Copyright © 2017 DigiPen Corporation

Example Without Virtual Destructor

```
#include <iostream>
class Base {
   public:
      Base(){ cout<<"Constructing Base";}
      ~Base(){ cout<<"Destroying Base";}
};
class Derive: public Base {
   public:
       Derive(){ cout<<"Constructing Derive";}
       ~Derive(){ cout<<"Destroying Derive";}
 };
void main() {
   Base *basePtr = new Derive();
   delete basePtr;
}
```

```
Constructing Base
Constructing Derive
Destroying Base
```

Copyright © 2017 DigiPen Corporation

Example With Virtual Destructor

```
#include <iostream>
class Base {
   public:
      Base(){ cout<<"Constructing Base";}
      virtual ~Base(){ cout<<"Destroying Base";}
};
class Derive: public Base {
   public:
       Derive(){ cout<<"Constructing Derive";}
       ~Derive(){ cout<<"Destroying Derive";}
 };
void main() {
   Base *basePtr = new Derive();
   delete basePtr;
}
```

```
Constructing Base
Constructing Derive
Destroying Derived
Destroying Base
```

Copyright © 2017 DigiPen Corporation

When Use Virtual Destructor?

- One important design paradigm of class design is that if a class has one or more virtual functions, then that class should also have a virtual destructor

Copyright © 2017 DigiPen Corporation

Conversions With Multiple Base Classes

- A pointer or reference to a derived class can be converted to a pointer or reference to any of its **public**! base classes
- Access protection (**protected** or **private**) prevented such conversions
- MS VS compiler output:

```
error C2243: 'type cast' : conversion from
'Panda *' to 'Endangered *' exists, but is
inaccessible
```

Copyright © 2017 DigiPen Corporation

Conversions With Multiple Base Classes (contd)

- Under multiple inheritance, there is a greater possibility of encountering an **ambiguous** conversion
- The compiler makes no attempt to distinguish between base classes in terms of a derived-class conversion
- Converting to each base class is equally good

Copyright © 2017 DigiPen Corporation

Access To Members Under Multiple Inheritance

DigiPen INSTITUTE OF TECHNOLOGY

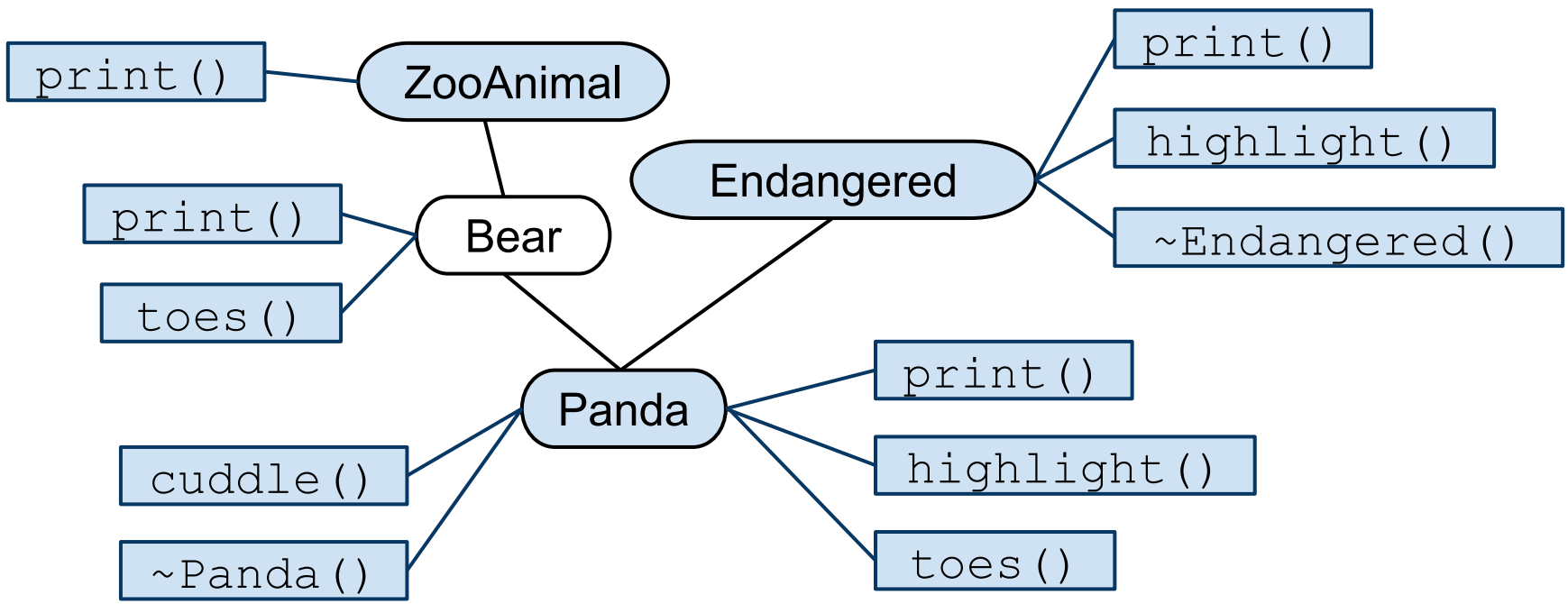
- Using a pointer to one base does not allow access to members of another base

- For example, assume that classes define following member functions:

  - `print()` - ZooAnimal, Bear, Endangered, Panda
  - `highlight()` - Endangered, Panda
  - `toes()` - Bear, Panda
  - `cuddle()` - Panda
  - `destructor` - Panda, Endangered

Copyright © 2017 DigiPen Corporation

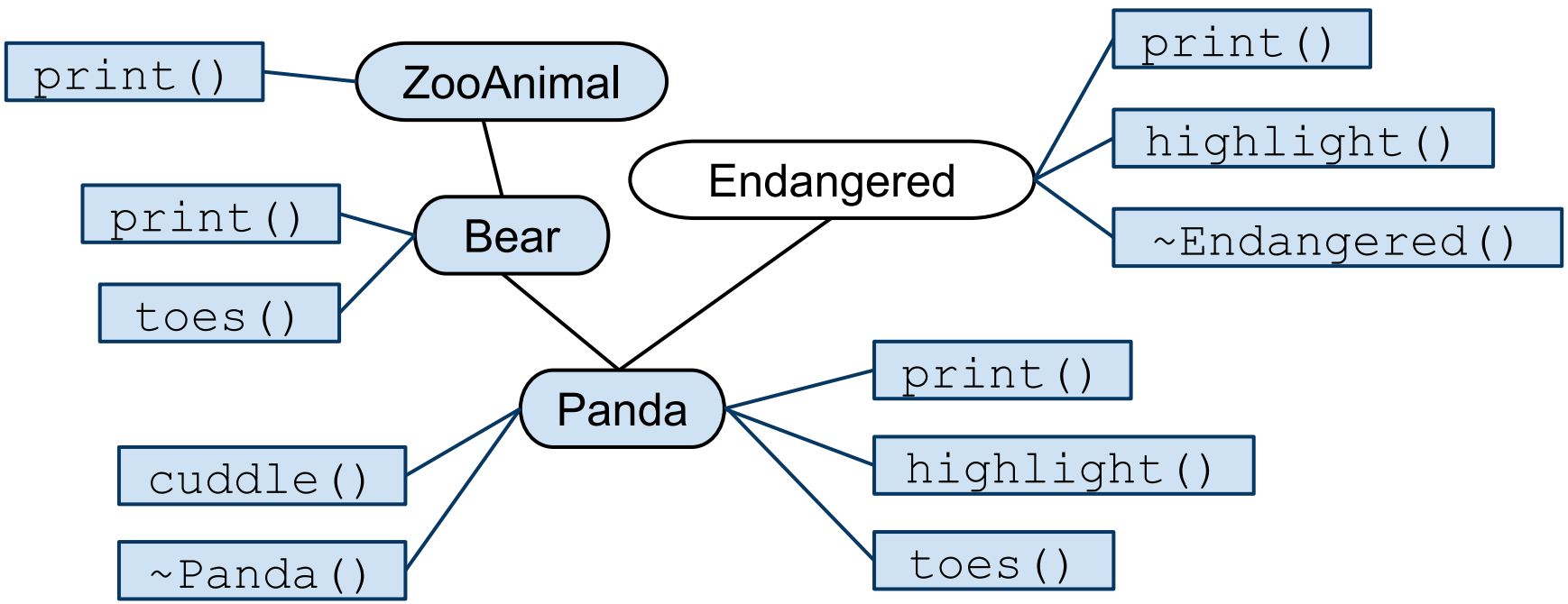
Access To Members Example




```
Bear *pb = new Panda();
pb->print(cout); // ok: Bear::print(ostream&)
pb->cuddle();    // err: not part of Bear
pb->highlight(); // err: not part of Bear
delete pb;       // ok: Bear::~Bear()
```

Copyright © 2017 DigiPen Corporation

Access To Members Example (contd




```
Endangered *pe = new Panda();
pe->print(cout); // ok: Endangered::print(ostream&)
pe->toes();      // error: not part Endangered
pe->cuddle();    // error: not part Endangered
pe->highlight(); // ok: Endangered::highlight()
delete pe;       // ok: ~Endangered()
```

Copyright © 2017 DigiPen Corporation

Class Scope Under Multiple Inheritance

- If the name is found in more than one base class,
  - then the use of that name must explicitly specify which base class to use with scope operator `::`
  - otherwise, the use of the name is ambiguous **even if they have different parameters list** or access level

- The best way to avoid potential ambiguities is to define a version of the function in the derived class that resolves the ambiguity

Copyright © 2017 DigiPen Corporation

Copy Control For Multiple Derived Classes

- The memberwise copy construction and assignment of a multiply derived classes behave in the same way as under single inheritance
- Each base class is implicitly copy constructed and assigned using that base class' own copy constructor, or assignment operator
- If the derived class defines its own copy constructor or assignment operator, then the class is responsible for copying (assigning) all the base class subparts
- The base parts are automatically copied or assigned only if the derived class uses the synthesized versions of these members

Copyright © 2017 DigiPen Corporation