Advanced Inheritance

C++ Object Oriented Programming

Pei-yih Ting
NTOU CS

Contents

Abstract Class

- In the University database program, Person class exists only to serve as a common base class
- We can strengthen the abstraction by allowing only derived objects of Person to be created (instantiated). Ex.

```
class Person {
public:
   Person();
   Person(char *name, int age);
   virtual ~Person();
   virtual void display() const = 0;
private:
   char *m_name;
   int *m_age;
};
```

At least one member function should be declared in such way for Person to be an abstract class

- Person is now an example of an abstract class. Any attempt to define a Person object will fail, i.e.

```
Person teacher; // compilation error
```

error C2259: 'Person' : cannot instantiate abstract class due to following members:
warning C4259: 'void __thiscall Person::display(void) const' :pure virtual function was not defined

What can you do with an Abstract Class?

- You can define a pointer to the abstract class object as long as you do not try to allocate an actual object (i.e. instantiation), e.g.

```
Person *ptrTeacher;            // polymorphic pointer
```

- Each of the derived class that need to be instantiated must implement its version of the display() virtual function. Otherwise, the derived class is still an abstract class and can not be instantiated.
- If Undergraduate, Graduate, and Faculty all implement the display(), function, then you can do this

```
Person *database[3];        // heterogeneous container
database[0] = new Undergraduate("Mary", 18);
database[1] = new Graduate("Angela", 25, 6000, "Fairview 2250");
database[2] = new Faculty("Sue", 34, "Fairview 2248", "Professor");
for (int i=0; i<3; i++)
   database[i]->display();
```

- Abstract classes are sometimes called *partial classes*

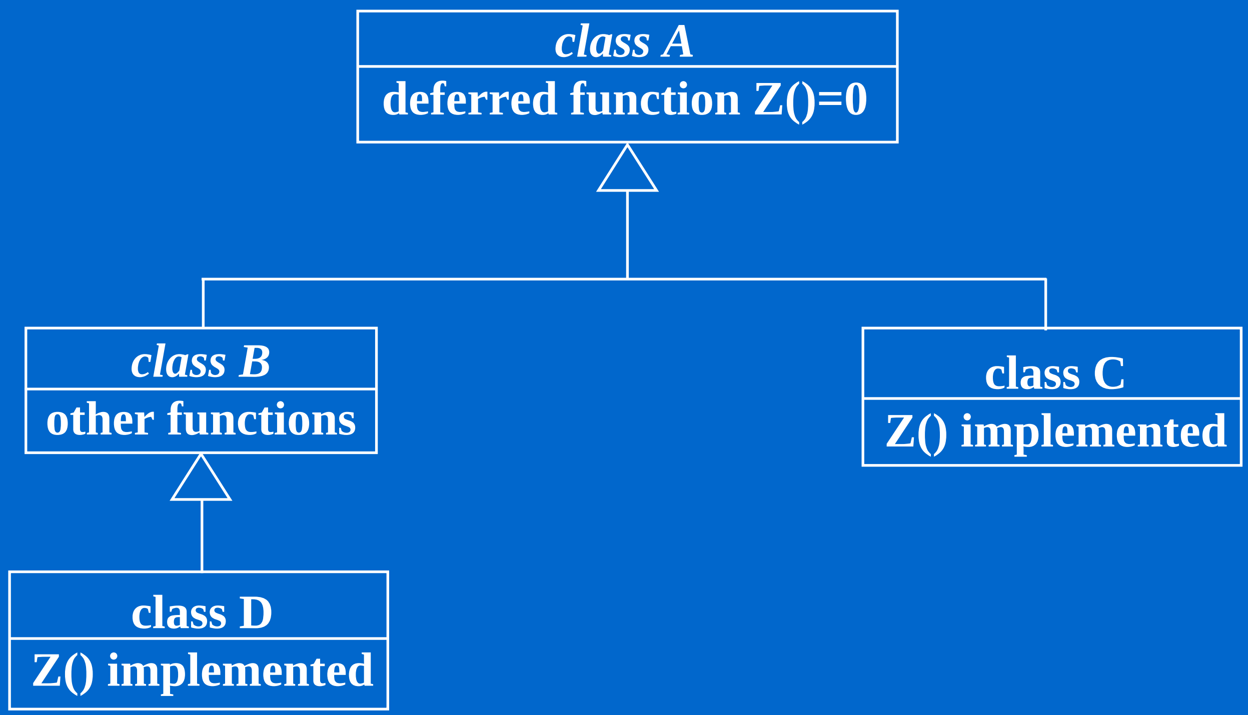
Pure Virtual Function

- The function that makes the class abstract is called a *pure virtual function* (also called a *deferred function*)
- The base class can define a version for this pure virtual function to be automatically shared by all derived classes. Since each derived class has to define its own implementation for this pure virtual function, the function defined will be overridden in all derived classes. However, this function can be called explicitly as follows:

```
void Person::display() const {
   cout << getName() << " is " << getAge() << " years old.\n";
}
void Faculty::display() const {
   Person::display();
   cout << " Her address is " << m_office.getAddress() << ".\n";
   cout << " Her rank is " << m_rank << ".\n\n";
}
```

Abstract Base Class (ABC)

- ABCs are base classes that contain some pure virtual functions without being implemented
- Ex. In the class hierarchy below, classes A and B are all abstract because function Z is not implemented till classes C and D



Why do you need Abstract Classes?

- There could be many roles a particular type of object is playing depending on which environment the object is in. For example,
  - A person is an employee in his office, a father in his family, a pitcher in a baseball game, etc
  - A pipe could be an output unit for one program and an input unit for another.
  - A printer could be an output device for a program and a resource to be handled by the operating system
- With abstract classes, you can describe multiple interfaces when viewing/using the object in different environments.
- An interface specifies a particular role (we specify a role with a set of operations) for an object that provides some particular functions to other objects. An ABC is frequently an adjective, Ex. Printable, Persistent, … only specify some properties.
- A class can have many unrelated abstract specifications. We will discuss this language feature in C++ as multiple inheritance.

Why do you need Abstract Classes?

- Two examples (types) of usage:

need not be instantiated

Student

Undergraduate

Graduate

ForeignStudent

Runnable

Printable

Observable

only describe some
partial property

WorkThread

Multiple Inheritance

- Sometimes an object has IS-A relationships to more than one class. In such cases, multiple inheritance may be appropriate.
- Consider the following two base classes

```
class Predator
{
public:
   Predator(char *prey, char *habitat);
   ~Predator();
   const char *getPrey() const;
   const char *getHabitat() const;
private:
   char *m_prey;
   char *m_habitat;
};
```

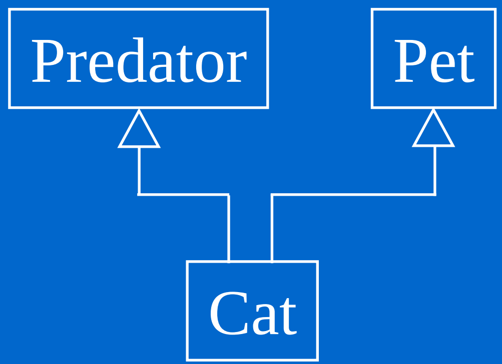
```
class Pet
{
public:
   Pet(char *name, char *habitat);
   ~Pet();
   const char *getName() const;
   const char *getHabitat() const;
private:
   char *m_name;
   char *m_habitat;
};
```

Multiple Inheritance (cont'd)

- Now we want to define a Cat class

```
class Cat: public Predator, public Pet
{
public:
   Cat(char *name, char *prey, char *habitat);
   void reduceLives();
   int getLives() const;
private:
   int m_lives;
};
```

- Class inheritance hierarchy



- The Cat constructor

```
Cat::Cat(char *name, char *prey, char *habitat)
   : Predator(prey, habitat), Pet(name, habitat), m_lives(9)
{
}
```

- Note that getHabitat() and the m_habitat will be inherited twice

Using the Multiple Inherited classes

- Using the Cat class

```
Cat cat("Binky", "mice", "indoors");
cat.reduceLives();   // due to an accident
cout << cat.getName() << " is a cat who eats " << cat.getPrey() << " and lives "
     << cat.Pet::getHabitat() << ".\n" << cat.getName() << " currently has "
     << cat.getLives() << " lives.\n";
```

Output
Binky is a cat who eats mice and lives indoors.
Binky currently has 8 lives

- What would happen if we wrote this?

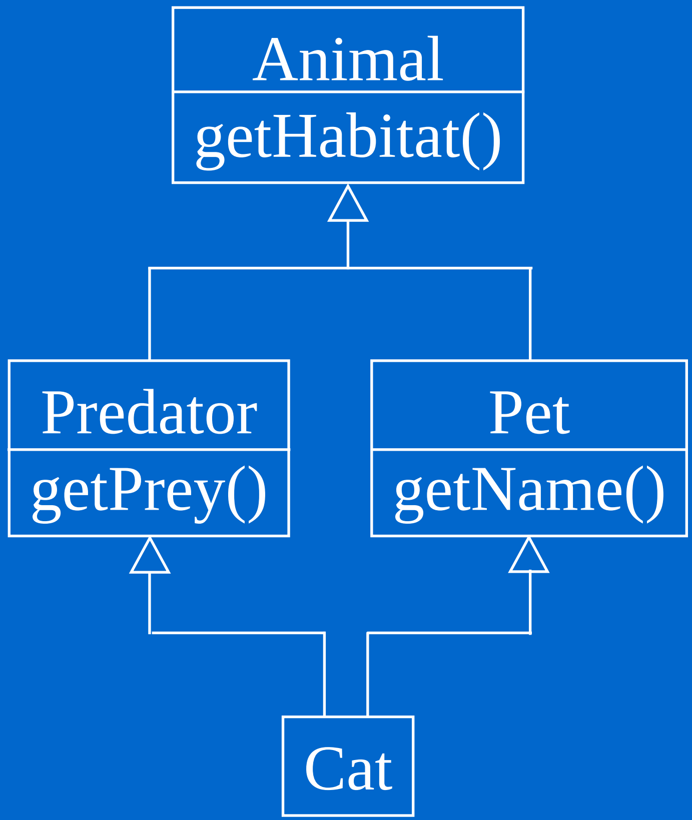
```
cout << cat.getHabitat();
```

error C2385: 'Cat::getHabitat' is ambiguous

- It is necessary to disambiguate which getHabitat() function we want. In this case, either Predator::getHabitat() or Pet::getHabitat() is a possible candidate.

Improving Multiple Inheritance

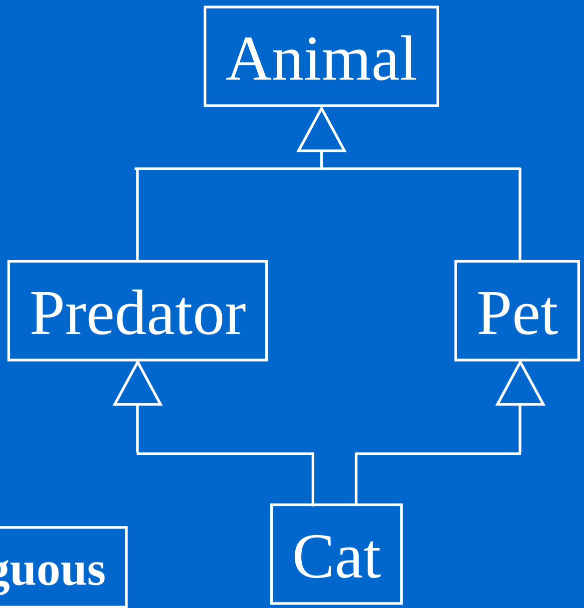
- The redundancy in the base classes is a clue that perhaps we haven't decomposed the inheritance properly
- Here is one solution:



- The base class declaration

```
class Animal {
public:
   Animal(char *habitat);
   virtual ~Animal();
   const char *getHabitat() const;
private:
   char *m_habitat;
};
```

Virtual Base Class



- Cat inherits getHabitat() through Predator but also through Pet
- Cat still has two getHabitat()'s

```
cout << cat.getHabitat();
```

```
error C2385: 'Cat::getHabitat' is ambiguous
```

- Still need to disambiguate these two versions

```
cout << cat.Perdator::getHabitat() << "\n";
cout << cat.Pet::getHabitat() << "\n";
```

- A better solution is to create a *virtual base class*.
  A virtual base class is included only once in all derived classes.
  In the case of Cat, all paths from Animal to Cat must be marked as virtual, but only once.

Syntax of Virtual Base Class

- Animal is declared as before, but Predator and Pet must be marked virtual

```
class Predator: public virtual Animal {
   …
};
class Pet: public virtual Animal {
   …
};
```

If not supplied, call to default ctor will be added

- Cat remains almost the same
- One critical difference: a virtual base class must be initialized by its most derived class (Cat in this case)

```
Cat::Cat(char *name, char *prey, char *habitat)
   : Animal(habitat), Predator(prey, habitat), Pet(name, habitat), m_lives(9) {
}
Predator::Predator(char *prey, char *habitat) : Animal(habitat) {
   m_prey = new char[strlen(prey)+1];
   m_habitat = new char[strlen(habitat)+1];
}
```

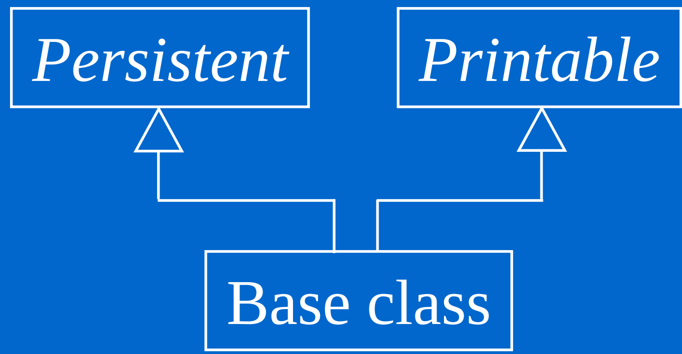
used only in Predator predator("a", "b");

- Any initialization from intermediate class is ignored.

Mix-in Inheritance

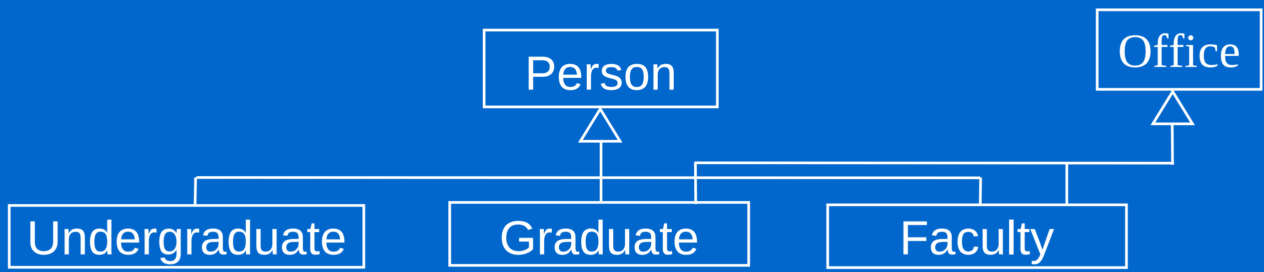
- Multiple inheritance is sometimes used to combine disparate classes into a single abstraction. This is called *mix-in inheritance*.

- Many class libraries combine classes so that all derived classes have access to key functionality. Ex.



The IS-A relationship is true only partially.

- The mix-in concept is easily abused, ex.



A graduate student is not an office definitely.

Private Inheritance

- Private inheritance

```
class Student {
public:
    Student();
    void setData(char *name, int age);
    int getAge() const;
    const char *getName() const;
private:
    char *m_name;
    int *m_age;
};
```

```
class Graduate: private Student {
public:
    Graduate(char *name, int age, int stipend);
    int display() const;
private:
    int m_stipend;
};
```

- All public members of Student are private to Graduate.
- Classes derived from Graduate would be unable to access any elements or services from Student.
- Private inheritance is equivalent to a HAS-A relationship. Outside client code cannot see any trace of the base class from a derived class object.

Restoring the Accessibility

- In private inheritance, individual functions can be restored to the original access (and only to that level).

```
class Student {
public:
    Student();
    void setData(char *name, int age);
    int getAge() const;
    const char *getName() const;
private:
    char *m_name;
    int *m_age;
};
```

```
class Graduate: private Student {
public:
    Graduate(char *name, int age, int stipend);
    int display() const;
    Student::getName;
private:
    int m_stipend;
};
```

- Usage

```
Graduate graduateStudent("Angela", 25, 6000);
cout << graduateStudent.getName();
```

Inherit from a Template Class

- Assume you have a templated array class

```
template <class type>
class Array {
public:
   Array(int arraySize);
   ~Array();
   void insertElement(int slot, type element);
   type getElement(int slot) const;
   int getSize() const;
private:
   int m_arraySize;
   type *m_array;
};
```

- You want the class to also return the largest element in the array

```
template <class type>
class NewArray: public Array<type> {
public:
   NewArray(int arraySize);
   type getLargest();
};
```

This derived NewArray class is still a template class.

Inherit from a Template Class

- Constructor

```
template<class type>
NewArray<type>::NewArray(int arraySize): Array<type>(arraySize) {
   for (int i=0; i<arraySize; i++) insertElement(i, 0);
}
```

- The new function

```
template<class type>
type NewArray<type>::getLargest() {
   type largest = getElement(0);
   for (int i=1; i<getSize(); i++)
      if (getElement(i) > largest)
         largest = getElement(i);
   return largest;
}
```

- Usage

```
void main() {
   NewArray<double> array(20);
   array.insertElement(0, 4.6);
   array.insertElement(5, 12.6);
   cout << array.getLargest();
}
```

Output
12.6