CSC340: Inheritance and Polymorphism

Main topics:

- Inheritance basics
- The slicing problem
- Polymorphism (late binding, virtual functions)
- Abstract classes/functions (pure virtual functions)

Readings:

- 5th edition: Chapter 8
- 6th edition: Ch1.4, Ch1.5, Ch2.4, and Interlude 4

Hui Yang
Computer Science Department
San Francisco State University
http://www.cs.sfsu.edu/~huiyang/

Why inheritance? An example.

- **Student vs. Person**
  - They have an IS-A relationship
    - A student is a person
- **Translate this to the OOP world**
  - Realize a class *Person* to contain and operate all the shared information (e.g., name, ssn)
  - Allow Student to inherit the shared information and methods

Basic Terminology

- **Syntax**

```
class Student : public Person
{
};
```

- **Student class will inherit all the data members and methods from Person**
- **Additional members can be declared within the Student class**
- **Person is a base class or super-class, and Student a derived class or sub-class**

 3

Inherited Members

- **A derived class inherits all the members of the parent class**
  - The derived class does not re-declare or re-define members inherited from the parent, **except**...
    - The derived class re-declares and re-defines member functions of the parent class that will have a different definition in the derived class
    - The derived class can add member variables and functions

 4

Private is Private

- A member variable (or function) that is private in the parent class is not directly accessible to the child class
- The parent class member functions must be used to access the private members of the parent
- This code would be illegal, since name is private to Person

```
void Student::print( )
 {
       cout << name <<endl;
         …
 }
```


Private, Protected and Public

- Public data and methods can be used by anyone
- Private data and methods can be used only by methods and friends of the class
- Protected data and methods can be used only by methods and friends of both the class and any derived class

Kinds of Inheritance

- **Public inheritance**
  - Public and protected members of the base class remain, respectively, public and protected members of the derived class
- **Protected inheritance**
  - Public and protected members of the base class are protected members of the derived class
- **Private inheritance**
  - Public and protected members of the base class are private members of the derived class
- **In all cases, the private section of a base class cannot be accessed by a derived class**

Derived Class Types

- **A Student object is (or contains) a Person object**
  - In C++, an object of type Student can be used where an object of type Person can be used
- **An object of a class type can be used wherever any of its ancestors can be used**
- **An ancestor cannot be used wherever one of its descendents can be used**

Redefining Member Functions

- **One can redefine a member function inherited from the base class in the derived class**
  - E.g., print() is defined in Person
  - It can be redefined in Student
- **Invoke a redefined function**

```
Student s;
s.print(); //call the version in Student
s.Person::print(); //call the version in Person
```


Default Constructor

- **If a derived class constructor does not invoke a base class constructor explicitly, the base class default constructor will be used**
- **If class B is derived from class A and class C is derived from class B**
  - When a object of class C is created
    - The base class A's constructor is the first invoked
    - Class B's constructor is invoked next
    - C's constructor completes execution

Constructors in a Derived Class

- A derived class often needs to include its own constructors
- The base class constructor can be invoked in the initialization section:

```
Student::Student(string name): Person(name), grade(1), hours( 0)
{
    //no code needed
}
```


Big-3 in Inheritance

- Inheritance has exceptions: some features in the base class cannot be inherited, but can be invoked.
  - Constructors
  - Destructor
  - Overloaded assignment operator

Default Behavior

- **The copy constructor:**
  - Invoking the copy constructor of the base class(es), followed by invoking copy constructors on the newly added members.
  - Will not work with pointers and dynamic variables
- **Assignment operator**
  - Invoking the default assignment operator of the base class(es), followed by invoking assignment operators on newly added data members
  - Will have nothing to do with the overloaded assignment operator in the base class, i.e., will not work with dynamic variables
- **Destructor**
  - Invoking destructors on each of the newly added data members, followed by invoking the destructor of the base classes

The Copy Constructor

- **Invoking the base class copy constructor sets up the inherited member variables**

```
Derived::Derived(const Derived& object) :Base(object), <other init>
{
...
}
```

- **Since object is of type Derived it is also of type Base**

The Operator = Implementation

- How to begin the implementation of the = operator for a derived class?

```
Derived& Derived::operator= (const Derived& rhs)
{
   Base::operator=(rhs);  //calling the base class's assignment
                          //operator to assign the inherited members
   ...
}
```


Destructors and Derived Classes

- The derived class should define its own destructor
- The derived class destructor need only use delete on dynamic variables added in the derived class, and data they may point to

The Slicing Problem

- **An example**

```
Student s_mary("mary", ... );
Person & p_mary = s_mary;
p_mary.print();
```

  - The last line will only print the Person contained within s_mary
  - Hence, the slicing problem

Solution to the Slicing Problem

- **Trigger late binding or dynamic dispatch by declaring print() as a virtual function in the Person class**

```
Class Person{
    ...
    virtual void print() const;
    ...
};
```

- **"virtual" will be inherited by derived classes, but can be overridden**
- **Runtime type check will be invoked to call the correct version**
- **Incurring overhead**

When to use virtual functions?

- **If a method might be expected to have a different implementation**
  - E.g., the print() in Person
- **Compared to the *final* keyword in Java**
  - Lack of "virtual" indicates that a function is final
- **Always declare a destructor as a virtual function**

```
Student *s1 = new Student(...);
Person *p1 = s1;
delete p1;
```


Abstract Classes in C++

- **A class is abstract if it includes a pure virtual function**

```
class Person{
        ...
        virtual void pure() const = 0;
        ...
};
```

- **An abstract class cannot be used to instantiate objects, therefore is only useful in the context of inheritance**

Using Polymorphism in C++

- **Example: Doodlebugs vs. Ants**
- **Classes: organism→doodlebugs and ants**
- **To manage a collection of organisms**
  - Java:
    - Declare an array of organism objects and will be automatically dispatched to manage doodlebug or ants
  - C++
    - Declare every method as virtual
    - Only allocate objects using "new"
    - Access all objects by pointers
    - Use an array of pointers
- **Demonstration**

21

Multiple Inheritance

```
class  Person
{
   public:
       …
  private:
       string name;
       int      ssn;
};
```

```
class  Employee: public Person
{
   public:
       …
  private:
       int      hours; //vacation hours
};
```

```
class  Student: public Person
{
   public:
       …
  private:
       int      hours; //credit hours
};
```

```
class  StudentEmployee: public Student,
public Employee
{
   public:
       …
  private:
       ???
};
```


22

Virtual Public Inheritance

```
class  Person
{
   public:
       …
  private:
       string name;
       int      ssn;
};
```

```
class  Employee: virtual public Person
{
   public:
       …
  private:
       int      hours;
};
```

```
class  Student: virtual public
Person
{
   public:
       …
  private:
       int      hours;
};
```

```
class  StudentEmployee: public Student,
public Employee
{
   public:
       …
  private:
       ???
};
```


Inheritance: Other Details

- Friendship is not inherited
- Protected and private inheritance
- The return type of an overridden function can be a subclass of the original return type
- C++ does not have interfaces, but can be achieved by declaring an abstract class

Summary

- **Inheritance**
  - Private is private
  - Overloading
  - Big-3
- **Polymorphism**
  - When and why?
  - Pointers & polymorphism
- **Comparison with Java**
  - Almost identical
  - But more challenging due to the need of using "pointers"

Quiz

- **A derived class can directly access the private member of its base class.**
- **A member function in a base class can be redefined in its derived class. The object of the derived class however can still access the base version of this member function.**
- **An abstract ADT is an ADT, of which at least one virtual member function is set to 0.**
- **One is strongly recommended to declare the destructor of a base class (when applicable) as a virtual function.**