1

Polymorphism and Inheritance

CSC 340

March 9, 2016

2

Overview

- **Inheritance Basics**
- Multiple Inheritance
- Polymorphism
- Miscellaneous Details

3

Inheritance Example

- Student vs. Person
  - They have an "is-a" relationship: a Student is a Person
- Translating into OOP
  - Create a class Person to encapsulate basic person attributes and behavior: name, ssn
  - Allow Student to inherit the Person information and methods, and build on it

Basic Terminology

- Syntax

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

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

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

Access Levels

- 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

Access Levels - Example

- This code would be illegal - Why?

```
class Person
{
  string name;
}
```

```
// student.h
class Student : public Person
{
  public:
    void print();
}

// student.cpp
void Student::print()
{
  cout << name << endl;
}
```

7

7

name is a private variable of Person (remember class declarations are private by default), so can not be directly accessed by Student

Access Levels

- 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

8

8

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

9

9

10

Derived class types

- A Student object is 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 whenever one of its descendants can be used

11

Derived class types

```
// Legal
Person[] people = new Person[30];
people[0] = new Student();

// Illegal
Student[] students = new Student[30];
students[0] = new Person();
```


12

Redefining Member Functions

- Programmers can re-define a member function inherited from the base class in the derived class

```
// person.cpp
void Person::print()
{
  cout << "Person: " << name << endl;
}

// student.cpp
void Student::print()
{
  cout << "Student: " << get_name() << endl;
}
```

Redefining Member Functions

- Invoke a re-defined function

```
Student s;

// to call the version in Student
s.print();

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

13

13

This implies that the Student object has a reference to it's parent type - we'll see this more in multiple inheritance

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 a class A, and class C is derived from class B
  - When an object of class C is created
    - The base class A's constructor is first invoked
    - Class B's constructor is invoked next
    - C's constructor completes execution

14

14

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 }
```

15

15

Some Exceptions

16

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

16

Default Behavior - Copy Constructor

17

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

```
Derived::Derived( const Derived& object) : Base (object)
{}
```

17

Review of copy constructor: copies state of one object of same type to new object (Why is object marked const?)
Derived first_thing;
Derived second_thing(first_thing);
Remember assignment defaults to single argument constructor, so this allows second_thing = first_thing

Why can we pass object, of type Derived, to Base class constructor? Derived is a Base.

Default Behavior - Assignment Operator

18

- 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 overloaded assignment operator in the base class (so will not work with dynamic variables)

```
Derived& Derived::operator =(const Derived& object)
{
  // First, call base class' assignment
  // operator to handle inherited members
  Base::operator=(rhs);

  // Now, handle Derived class assignment
}
```

18

Default Behavior - Destructor

- Invoking destructors on each of the newly added data members, followed by invoking the destructor of the base class
- The derived class should define its own constructor
- The derived class destructor need only use delete on dynamic variables added in the derived class, and data they may point to

19

19

More on delete later, but it's dynamic memory management

Overview

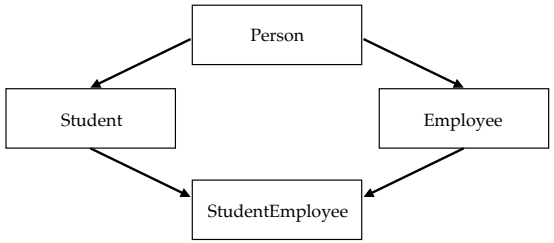
- Inheritance Basics
- **Multiple Inheritance**
- Polymorphism
- Miscellaneous Details

20

20

Multiple Inheritance

- Classes can be derived from multiple base classes



21

21

Multiple Inheritance

- Example

```
class Person {};
class Student : public Person {};
class Employee : public Person {};
class StudentEmployee : public Student, public Employee {};
```


22

Multiple Inheritance

- This can lead to ambiguity - what happens if a method print, in Person is called in StudentEmployee?
  - How does the run time look up the appropriate implementation? Is it Student::Person.print(), or Employee::Person.print()?

23

StudentEmployee has two possible base class references to Person:

1. Employee::Person
2. Student::Person

Multiple Inheritance

- We can remove this ambiguity through the use of virtual class inheritance

```
class Person {};
class Student : public virtual Person {};
class Employee : public virtual Person {};
class StudentEmployee : public Student, public Employee {};
```


24

Overview

- Inheritance Basics
- Multiple Inheritance
- **Polymorphism**
- Miscellaneous Details

25

Polymorphism

- Using pointers (or references), C++ can look up the type of a variable at run time
  - Late binding (or dynamic binding) - the definition of a method is not bound to an object (resolved) until runtime
  - Early binding (or static binding) - the compilation phase fixes all types of variables and expressions

26

Many shapes - one thing can behave in many different ways

The Slicing Problem

- When you assign an object of a derived class to an instance of a base class, losing part of the information
- Example

```
class A {
  int a_variable;
};

class B : public A {
  int b_variable;
};

B b;
A a = b; // Loses b_variable (b_variable is "sliced")
```


27

The Slicing Problem

28

- Another example: https://gist.github.com/jrob8577/c896f91ca8db8d038639
- The last line will only print the Person contained within mary

Solution to Slicing Problem

29

- 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 (late/dynamic binding)
- Adds overhead

When to use virtual functions?

30

- If a method might be expected to have a different implementation in derived classes
  Example: https://gist.github.com/jrob8577/0e1fce58c82cabf293cb
- Compared to the final keyword in Java - the lack of "virtual" indicates that a function is final (can not be overridden)
- 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 a_method() const = 0;
};
```

- The = 0 portion of that statement indicates it's "pure"
- An abstract class cannot be used to instantiate objects, therefore it is only useful in the context of inheritance

31

31

Why is this useful?

Overview

- Inheritance Basics
- Multiple Inheritance
- Polymorphism
- **Miscellaneous Details**

32

32

Miscellaneous Details

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

33

33