CS 162
Intro to Programming II

Inheritance I
Aggregation and Composition

• **Class aggregation**: An object of one class owns an object of another class

• **Class composition**: A form of aggregation where the enclosing class controls the lifetime of the objects of the enclosed class

• Supports the modeling of ‘has-a’ relationship between classes – enclosing class ‘has a(n)’ instance of the enclosed class
Object Composition

class StudentInfo
{
    private:
        string firstName, LastName;
        string address, city, state, zip;
    ...
};

class Student
{
    private:
        StudentInfo personalData;
    ...
};
Member Initialization Lists

- Used in constructors for classes involved in aggregation.
- Allows constructor for enclosing class to pass arguments to the constructor of the enclosed class.
- Notation:

```cpp
owner_class(parameters):owned_class(parameters);
```
Member Initialization Lists

Use:

class StudentInfo
{
    ...
};
class Student
{
    private:
        StudentInfo personalData;
    public:
        Student(string fname, lname):
            StudentInfo(fname, lname);
};
Member Initialization Lists

• Member Initialization lists can be used to simplify the coding of constructors
• Should keep the entries in the initialization list in the same order as they are declared in the class
Aggregation Through Pointers

• A ‘has-a’ relationship can be implemented by owning a pointer to an object

• Can be used when multiple objects of a class may ‘have’ the same attribute for a member
  – ex: students who may have the same city/state/zipcode

• Using pointers minimizes data duplication and saves space
Aggregation, Composition, and Object Lifetimes

• Aggregation represents the owner/owned relationship between objects.
• Composition is a form of aggregation in which the lifetime of the owned object is the same as that of the owner object.
• Owned object is usually created as part of the owning object’s constructor, destroyed as part of owning object’s destructor.
Inheritance

- **Inheritance** is a way of creating a new class by starting with an existing class and adding new members.
- The new class can replace or extend the functionality of the existing class.
- Inheritance models the 'is-a' relationship between classes.
Inheritance - Terminology

• The existing class is called the **base class**
  – Alternates: **parent class, superclass**

• The new class is called the **derived class**
  – Alternates: **child class, subclass**
Inheritance Syntax and Notation

// Existing class
class Base
{
};

// Derived class
class Derived : public Base
{
};
Inheritance of Members

class Parent
{
   int a;
   void bf();
};
class Child : public Parent
{
   int c;
   void df();
};

Objects of Parent have members
   int a; void bf();

Objects of Child have members
   int a; void bf();
   int c; void df();
Protected Members and Class Access

• **protected member access specification**: A class member labeled `protected` is accessible to member functions of derived classes as well as to member functions of the same class.

• Like `private`, except accessible to members functions of derived classes.
Base Class Access Specification

Base class access specification determines how private, protected, and public members of base class can be accessed by derived classes.
C++ supports three inheritance modes, also called base class access modes:

- public inheritance
  ```cpp
class Child : public Parent { };
```
- protected inheritance
  ```cpp
class Child : protected Parent{ };
```
- private inheritance
  ```cpp
class Child : private Parent{ };
```
Base Class Access vs. Member Access Specification

Base class access is not the same as member access specification:

- Base class access: determine access for inherited members
- Member access specification: determine access for members defined in the class
Member Access Specification

Specified using the keywords private, protected, public

class MyClass
{
    private: int a;
    protected: int b; void fun();
    public: void fun2();
};
Base Class Access Specification

class Child : public Parent
{
    protected: base access
        int a;
    public: member access
        Child();
};
Base Class Access Specifiers

1) **public** – object of derived class can be treated as object of base class (not vice-versa)

2) **protected** – more restrictive than **public**, but allows derived classes to know some of the details of parents

3) **private** – prevents objects of derived class from being treated as objects of base class.
Effect of Base Access

Base class members

private: x  
protected: y  
public: z  

protected base class

private: y  
private: z  

public class members

How base class members appear in derived class

private base class

x inaccessible
private: y  
private: z  

protected base class

x inaccessible
protected: y  
protected: z  

public base class

x inaccessible
protected: y  
protected: z  

public: z  

Constructors, Destructors and Inheritance

• By inheriting every member of the base class, a derived class object contains a base class object

• The derived class constructor can specify which base class constructor should be used to initialize the base class object
Order of Execution

- When an object of a derived class is created, the base class’s constructor is executed first, followed by the derived class’s constructor.
- When an object of a derived class is destroyed, its destructor is called first, then that of the base class.
Order of Execution

// Student - base class
// UnderGrad - derived class
// Both have constructors, destructors

int main()
{
    UnderGrad u1;
    ...
    return 0;
}// end main

Execute Student constructor, then execute UnderGrad constructor

Execute UnderGrad destructor, then execute Student destructor
Passing Arguments to Base Class Constructor

- Allows selection between multiple base class constructors
- Specify arguments to base constructor on derived constructor heading
- Can also be done with inline constructors
- Must be done if base class has no default constructor
Passing Arguments to Base Class Constructor

class Parent {
    int x, y;
    public: Parent(int, int);
};
class Child : public Parent {
    int z
    public:
    Child(int a): Parent(a, a*a) {
        z = a;
    }
};
Overriding Base Class Functions

- **Overriding**: function in a derived class that has the *same name and parameter list* as a function in the base class.
- Typically used to replace a function in base class with different actions in derived class.
- Not the same as overloading – with overloading, the parameter lists must be different.
Access to Overridden Function

• When a function is overridden, all objects of derived class use the overriding function.

• If necessary to access the overridden version of the function, it can be done using the scope resolution operator with the name of the base class and the name of the function:

  \texttt{Student::getName();}