12 - Inheritance

November 1, 2022

COMP2404

Darryl Hill

Inheritance

Contents

1. Overview
2. Member Access
3. Constructors and Destructors
4. Types of Inheritance
5. Multiple Inheritance

Overview

Role of inheritance in object oriented design

- another method of abstraction, encapsulation

Classes can be a more detailed specification of another class

- ”is-a” relationship
- helps us reuse existing code

C++ terminology

- base class
- derived class

Inheritance

```
class Boat {
     public:
          void move(Direction&);

     private:
          int capacity;
};

class Sailboat: public Boat { };
```

- `Boat` is the base class
- `Sailboat` is the derived class

Member Access

Accessing base class members from derived class:

- all base class members are inherited
  - all data members
  - all member functions
- only `public` and `protected` base class members are accessible to the derived class
- `private` members are included in the derived class
  - but still `private` to base class
  - not directly accessible (unless derived class is a `Friend`)
  - still there! Memory is allocated and that memory contains a value.

coding example <p1>

Member Access

Summary:

- `public` and `protected` members can be accessed by the derived class
- `private` base class members can only be accessed through `public` or `protected` member functions
  - or `friend` classes and `friend` functions.
- never directly

Member Access

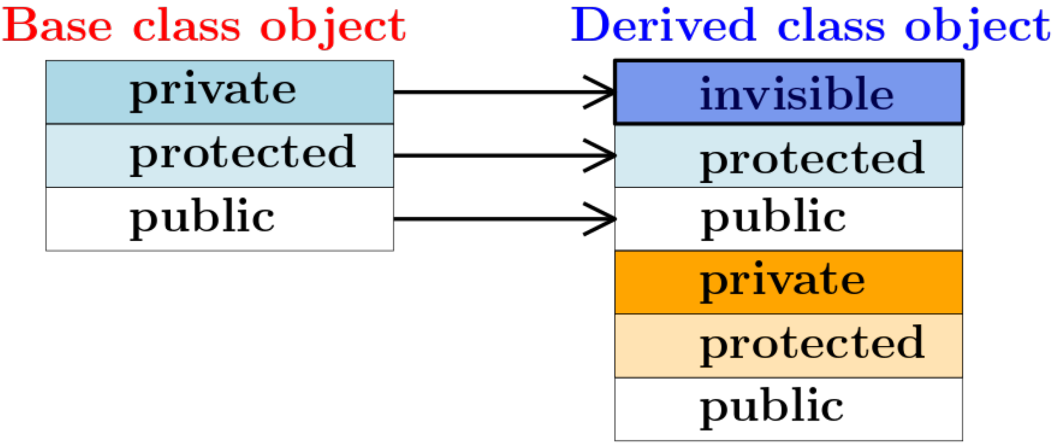
If we override a base class function we may still access it.

- Using the scope resolution operator.
- This means we can access overridden functions from anywhere up the class hierarchy
  - Java only has parent access using `super`

```
void Chicken::print() const
{
    Animal::print();

    cout << " and I'm a chicken that can produce " << eggCount
    << " eggs daily" << endl;
}
```

Member Access



Base class object

private

protected

public

Derived class object

invisible

protected

public

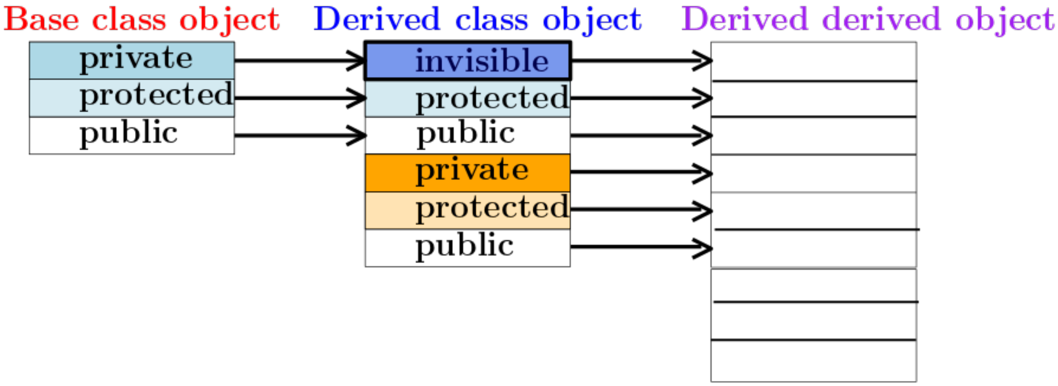
private

protected

public

Member Access

What happens here?

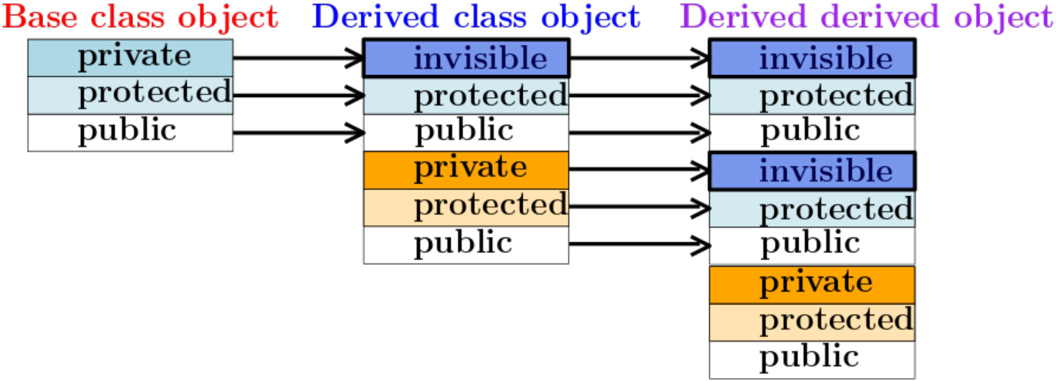


Member Access

COMP 2404 9

What happens here?

- private → invisible
- protected → protected
- public → public



Base class object
private
protected
public

Derived class object
invisible
protected
public
private
protected
public

Derived derived object
invisible
protected
public
invisible
protected
public
private
protected
public

Member Usage

Using inherited members

- public and protected accessed directly by name
- private members
  - accessed with public or protected member functions
    - data members or private helper functions
  - accessed with base class friend classes and friend functions

Member Usage

Overriding class members

- use scope resolution operator to access base member
- overriding member redefines and ***hides*** inherited member

Friendship

- a base class's friend functions and classes are ***not*** inherited

coding example <p2>

Coding Example

If we redefine a function of the base class in the derived class then

- The base class cannot use child class function
- The child class cannot use the derived class function directly - its hidden
  - Even if the signature is different
  - We can however access it using the scope resolution operator.

```
class Animal{
     void run(int distance);
};
class Cheetah: public Animal{
     void run(float speed);
};
```

Coding Example

```
class Animal
{
   public:
     void run(int distance);
};

class Hyena: public Animal
{
   public:
     void run(float speed);
};

Hyena banzai;
```

Given these class definitions we could not run

```
banzai.run(10);
```

Since the new declaration hides the old. But we could run

```
banzai.Animal::run(10);
```

Constructors and Destructors

Our new derived class has base class parts that need to be initialized.

Initializing base members directly in the derived class violates encapsulation.

- Also if they are `private` we have no access.
- How do we make sure they are properly initialized?

Base class constructor knows how to initialize.

We never override a constructor in C++.

- Each class in the hierarchy uses their own constructor.
- These constructors are responsible for initializing all member variables at that level.

Constructors and Destructors

When we initialize a derived class object in C++:

- Base class constructor is ***always*** called.
  - either explicitly, or
  - implicitly by calling default constructor.

- Problem: how do we explicitly call the base class constructor to pass in arguments?
  - Similar problem to initializing members, and has similar solution

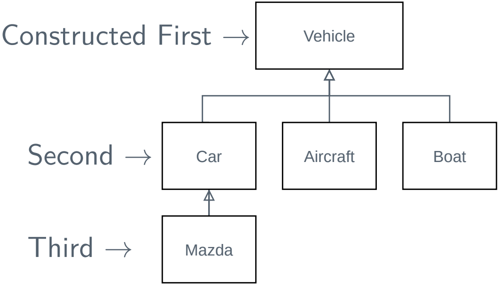
Constructors and Destructors

Order of invocation of constructors when using inheritance:

- constructors - built top down
  - base class part constructed first
  - derived class part constructed last

```
Mazda m;
```

Use ***base class initializer syntax*** to explicitly call base class constructors.



Constructed First → Vehicle

Second → Car

Aircraft

Boat

Third → Mazda

Base Class Initializer Syntax

```
class Animal {
    public:
        Animal(string n): name(name){}
    private:
        string name;
};

class Coyote: public Animal {
    public:
        Coyote(string name, float range): Animal(name),
        range(range){}
    private:
        int range;
};
```

We explicitly call base class constructor

Constructors and Destructors

Order of invocation of destructors when using inheritance:

- destructors - destroyed bottom up
  - derived class part destroyed first
  - base class destroyed last

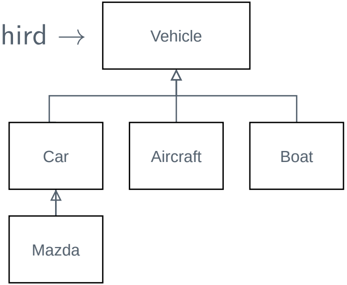
```
int main(){
    Mazda m;
    return 0;
}
```

When `main` exits, the `Mazda` is destroyed.

Third →

Second →

Destroyed First →



Destructors are always called implicitly, never explicitly.

Constructors and Destructors

Order of invocation - inheritance and composition

- constructors - top down, inside out
  1) base class
     a) containee constructors
     b) base class constructors
  2) derived class
     a) containee constructors
     b) derived class constructors

- destructors are reverse order of constructors

coding example <p3>

Types of Inheritance

C++ has three different types of inheritance.

Public inheritance

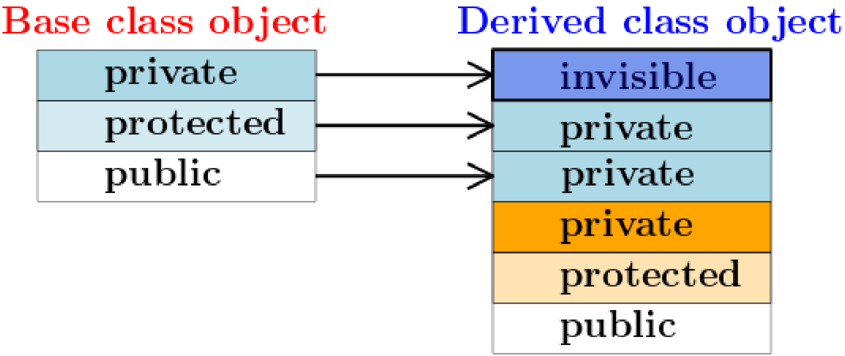
- this gives us the derived ’is-a’ base relationship
- this is ”inheritance” as we know it

Private or Protected inheritance

- not technically an ’is-a’ relationship
  - though all members are still inherited
- Used as substitute for *composition* and *delegation* to another object

Private Inheritance

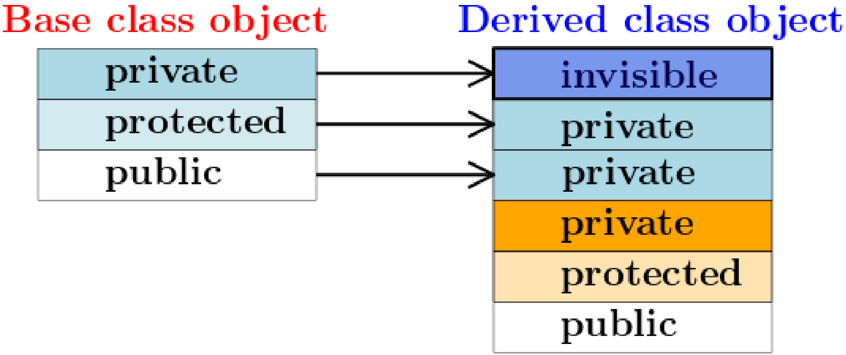
What happens here?



Private Inheritance

What happens here?

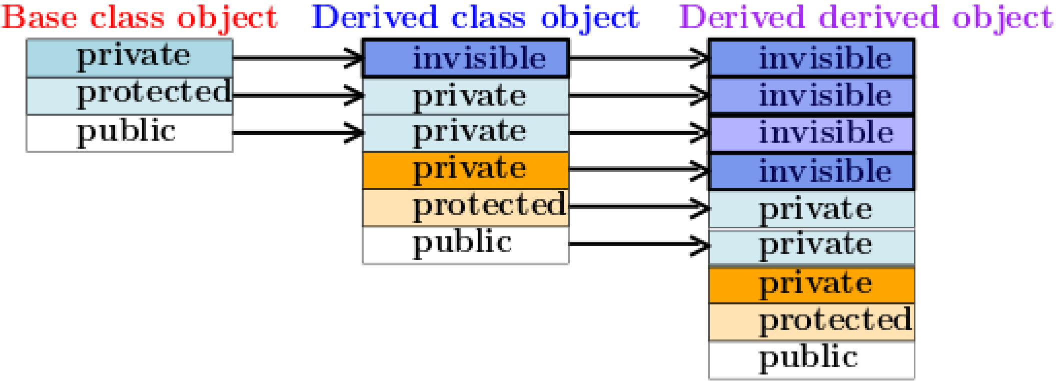
- private → invisible
- protected → private
- public → private



Private Inheritance

What happens here?

- private → invisible→ invisible
- protected → private→ invisible
- public → private→ invisible



Base class object

private
protected
public

Derived class object

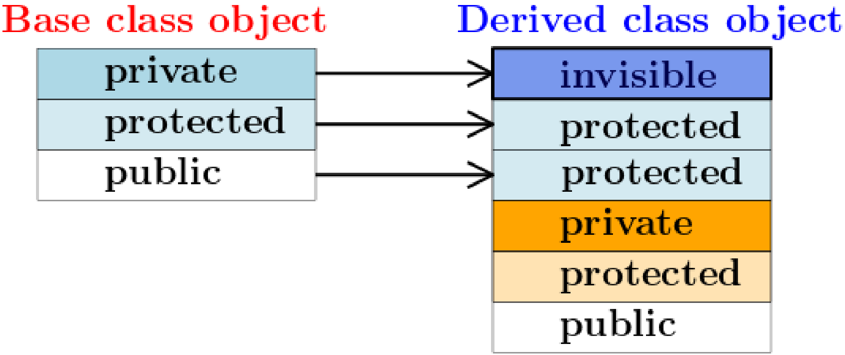
invisible
private
private
private
protected
public

Derived derived object

invisible
invisible
invisible
invisible
private
private
private
protected
public

Protected Inheritance

What happens here?



Base class object

private

protected

public

Derived class object

invisible

protected

protected

private

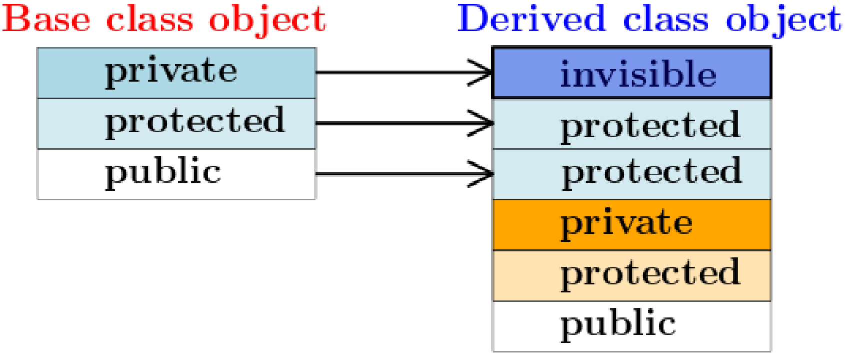
protected

public

Protected Inheritance

What happens here?

- private → invisible
- protected → protected
- public → protected



Base class object

private

protected

public

Derived class object

invisible

protected

protected

private

protected

public

Private and Protected

Not an "is-a" relationship.

Observe the code. An `Animal` should be able to `speak()`

- because we used private inheritance, `speak()` becomes private
- A `Goose` object cannot `speak()`
  - (at least not publicly).
- Therefore a `Goose` is not an `Animal`.
  - It does not have the same public interface.

```
class Animal{
    public:
        void speak();
};

class Goose: private Animal{};

Animal* a = new Goose;
```

This code would cause an error.

Why Private and Protected?

If it is not actually inheritance, then why `private` or `protected`?

- We can inherit privately and expose only those functions we want to expose.
- More efficient than delegation

Say we wanted a `Stack` class by leveraging the `vector` class

- there are `vector` functions we ***don't*** want called
- How do we handle this?
  - Use public inheritance and override unwanted functions with empty bodies
    - messy
    - functions can still be called - the public interface is misleading
  - Use composition and delegation
    - we choose the public interface
    - function calls must be forwarded to another object - not as efficient
  - Use private inheritance - we choose what is exposed and what is hidden

Example <p4>

Multiple Inheritance

When a class inherits from more than one base class

- Not directly supported in many OO languages BUT
  - Languages like Java simulate it using Interfaces
  - We can also simulate Java interfaces using multiple inheritance with abstract classes.
- Ambiguity is resolved using scope resolution operator

Coding example <p5>

Coding Example p5

Notes on Example p5:

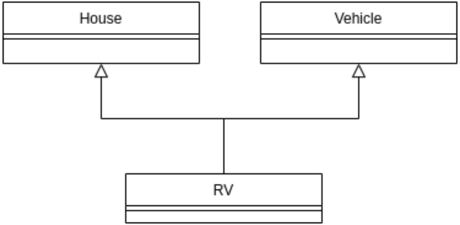
- If we have two of the same member variables
  - use scope resolution operator
    - (in fact, we must use it)
- If we use the virtual keyword, then we must call the super-super class constructor

Multiple Inheritance

Types of multiple inheritance

- distinct base class
- multiple inclusion base class
- virtual base class

Problem: diamond hierarchy

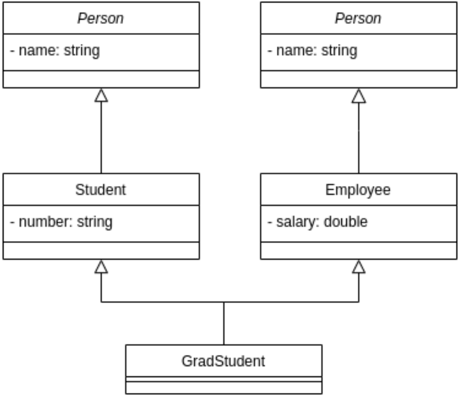


Multiple Inheritance

Types of multiple inheritance

- distinct base class
- multiple inclusion base class
- virtual base class

Problem: diamond hierarchy

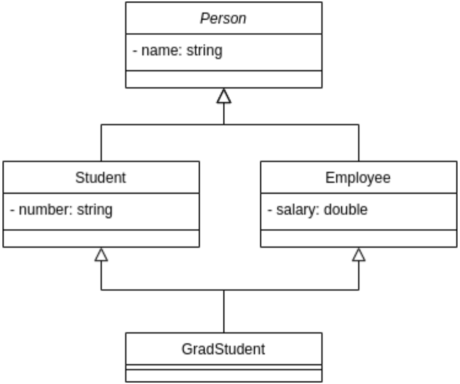


Multiple Inheritance

Types of multiple inheritance

- distinct base class
- multiple inclusion base class
- virtual base class

Problem: diamond hierarchy


Person
- name: string
Student
- number: string
Employee
- salary: double
GradStudent