Inheritance

CS1004 – Object Oriented Programming

Introduction

- Inheritance is probably most powerful feature of OOP
- Using Inheritance we create new classes from

existing classes

Inheritance

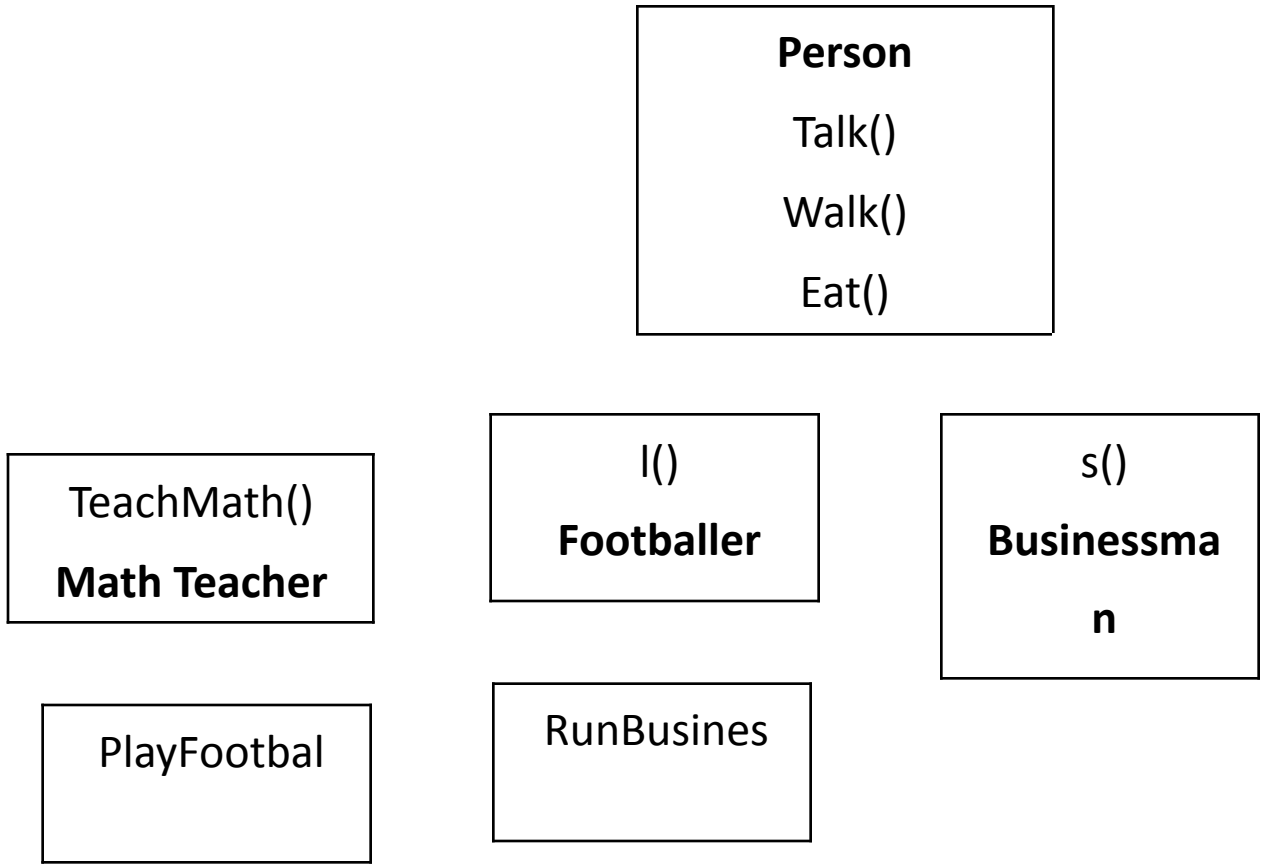
- Reusability is an important design goal of software engineering •
  Saves time and effort
  - Improves program structure and reliability

- Inheritance
  - provides us a mechanism of software reusability which is one of the most important principles of software engineering
  - The process of creating new classes, called derived classes, from existing classes or base classes creating a hierarchy of parent classes and child classes.
  - The child classes inherit the attributes and behaviour of the parent classes.

Inheritance in classes

- If a class '**Math Teacher**' inherits from class **Person**, then '**Math Teacher**' contains all the characteristics (information structure and behavior) of class **Person** • The parent(Person) class is called **base class** and the child class(Math Teacher) is called **derived class**
- Besides inherited characteristics, derived class may have its own unique characteristics, such as 'Teaching Mathematics'

- or over-ride inherited behaviour.



UML Notation

- Child class is derived from Parent class

- Derived Class is derived from Base Class

Class

Parent

Base Class

Class Child

Derived

Some definitions in class hierarchy

- Direct base class
  - Inherited explicitly (one level up hierarchy)
- Indirect base class
  - Inherited two or more levels up hierarchy
- Single inheritance
  - Inherits from one base class

- Multiple inheritance
  - Inheritance from multiple classes

Animals: Class's hierarchy

People

Mammal

Classification

Dog

Animal

Reptiles . . . . .

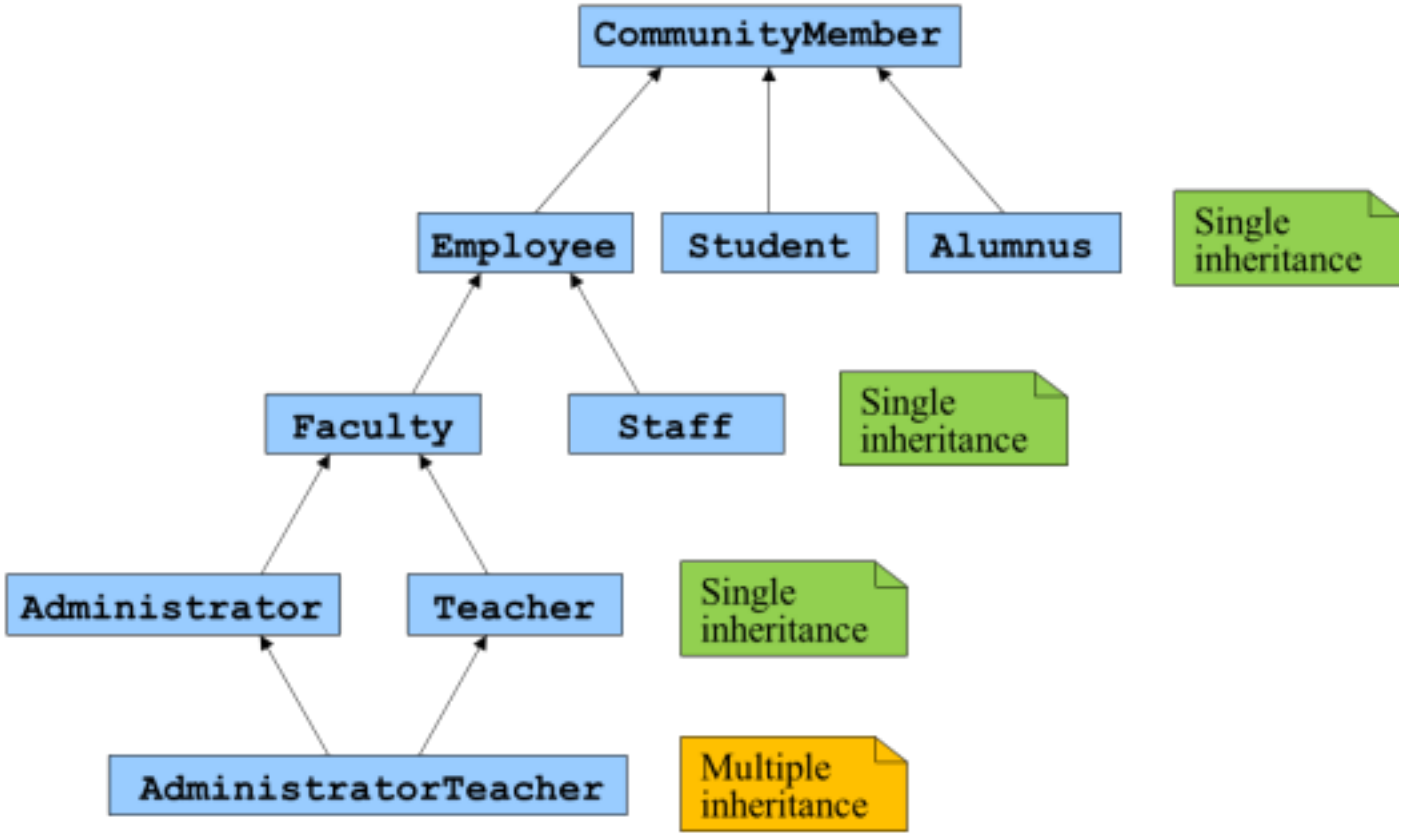
man woman John Mary

Inheritance

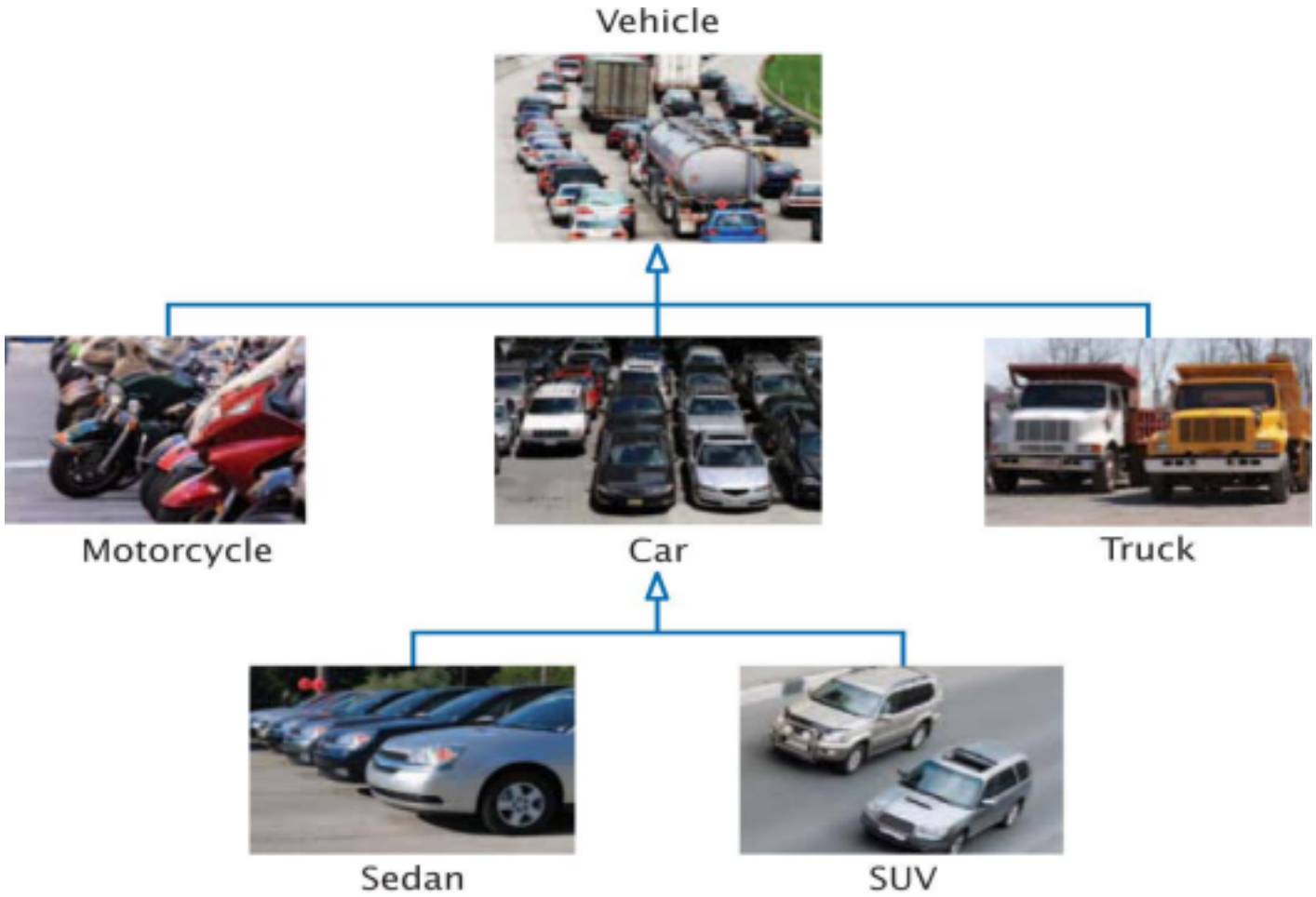
Inheritance Examples

Student	GraduateStudent UndergraduateStudent
Shape	Circle Triangle Rectangle
Loan	CarLoan HomeImprovementLoan
Employee	StaffMember FacultyMember
Account	CheckingAccount SavingAccount

Another example: University's community member's hierarchy



Inheritance Hierarchies


Vehicle
Motorcycle
Car
Truck
Sedan
SUV

Syntax for using Public Inheritance

- Public inheritance creates the “is-a” relationship
- every rectangle *is a* shape
- every lion *is an* animal
- every lawyer *is an* employee
- We say that an object of the derived class “is-a” object of the base class
- Syntax:

class *DerivedClassName* : access-level *BaseClassName*

where

- access-level specifies the type of derivation (inheritance)
- private by default, protected or public

- Most commonly used type of inheritance

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

Derived class Base class

Note: if the type of inheritance is omitted it defaults to ***private***!

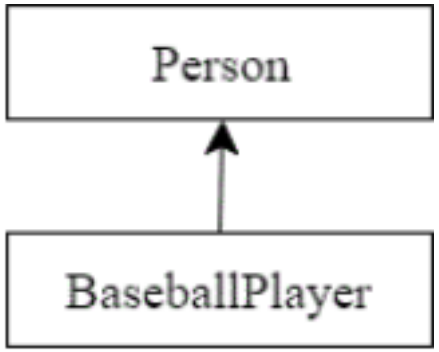
Inheritance in C++

- There are three types of inheritance in C++
  - Public
  - Private
  - Protected
- Public inheritance is commonly used form of inheritance.
- Protected and Private inheritance are rarely used.

Example: A Person Class

- Here's a simple class to represent a generic base class named person and a BaseballPlayer class which is derived from the Person Class.
- Every person (regardless of gender, profession, etc…) has a name and age, so those are

represented in the Person Class.



Example: A Person Class

- Every person (regardless of gender, profession, etc…) has a name and age, so those are represented here.

```
#include <string>

class Person {
```

```
// In this example, we're making our members public for simplicity
public:
     string m_name;
     int m_age;

     Person(string name = "", int age = 0)
          : m_name( name ), m_age( age )
     {
     }
  string getName() { return m_name; }
     int getAge() { return m_age; }
  };
```

Example (cont'd): A Baseball Player

- Baseball players need to contain information that is specific to baseball players -- for example:
- we might want to store a player's *batting average*, and the number of ***runs*** they've hit.

```
class BaseballPlayer {
// In this example, we're making our members public for simplicity
public:
```

```
    double m_battingAverage;
    int m_homeRuns;

  BaseballPlayer(double battingAverage = 0.0, int homeRuns =
0):
    m_battingAverage(battingAverage),
    m_homeRuns(homeRuns) {
    }
};
```

How to keep track of a baseball player's *name* and *age*, and while we already have that information as part of our Person class?

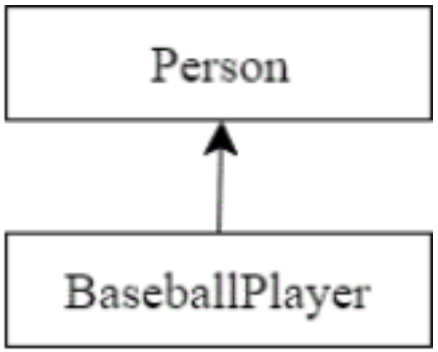
How to keep track of a baseball player's name and age?

- We have following choices for how to add ***name*** and ***age*** to BaseballPlayer:

1. Option 1: Add ***name*** and ***age*** to the BaseballPlayer class directly as members. This is probably the worst choice, as we're duplicating code that already exists in

our Person class.

2. Option 2: Have BaseballPlayer inherit those attributes from Person. Remember that inheritance represents an ***is-a*** relationship.
   - Is a BaseballPlayer a Person? **Yes**, it is. So inheritance is a good choice here.



Making BaseballPlayer a derived class

- Derived class BaseballPlayer acquires the member functions and variables from Class Person.
- Additionally, BaseballPlayer defines two members of its own: ***m_battingAverage*** and ***m_homeRuns***, since these properties are specific to a BaseballPlayer, not to any

Person.

```
class BaseballPlayer : Public Person
{
public:
double m_battingAverage;
int m_homeRuns;

BaseballPlayer(double battingAverage = 0.0, int
homeR uns = 0)
        : m_battingAverage(battingAverage),
m_homeRuns (homeRuns)
    {
    }
};
```

Creating Derived class object

- BaseballPlayer objects will have 4 member variables:
- ***m_battingAverage*** and ***m_homeRuns*** from BaseballPlayer, and ***m_name*** and ***m_age*** from Person.

```
int main() {
    // Create a new BaseballPlayer object
     BaseballPlayer harry;
// Assign it a name (we can do this directly because m_name is public)
harry.m_name = "Harry";
// Print out the name
cout << harry.getName() <<endl;  // use the getName() function we've acquired from the Person base class

 //this outputs: Harry
     return 0;
}
```

Public Inheritance

- While implementing the BaseballPlayer class we used the public inheritance:
  `class BaseballPlayer : Public Person`
- Using public Inheritance

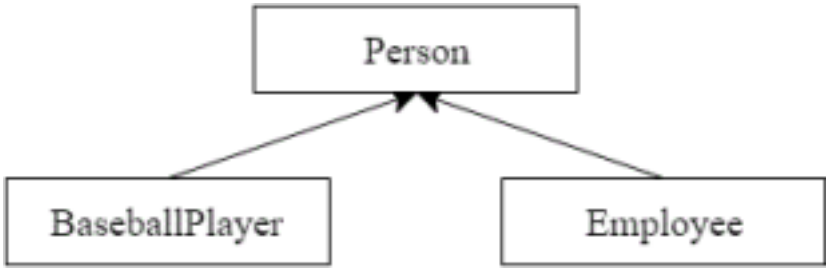
- the public members of the base class (Person) are also public in the derived class (BaseballPlayer)
- Private members of base class are not accessible from outside of base class, even in the derived class (Information Hiding)
- private members can be accessed using the member functions of the derived class.

- For example, in our previous example, Person's data members ***name*** and ***age*** are accessible by the derived class object.

An Employee Derived Class

- Now we derive a second class named Employee from the Person class.
- An employee **"is-a"** person, so using inheritance is appropriate:

Employee inherits ***m_name*** and ***m_age*** from Person (as well as the two access functions), and adds two more member variables and a member function of its own.



An Employee Derived Class

- We derive an Employee class from Person. An employee "is-a" person, so using inheritance is appropriate.
  - Employee class has two data members and one member function.

```
// Employee publicly inherits from Person
class Employee: public Person
{ public:
    double m_hourlySalary;
```

Employee inherits m_name and m_age from Person (as well as the two access functions), and adds two more member variables and a member function of its own.

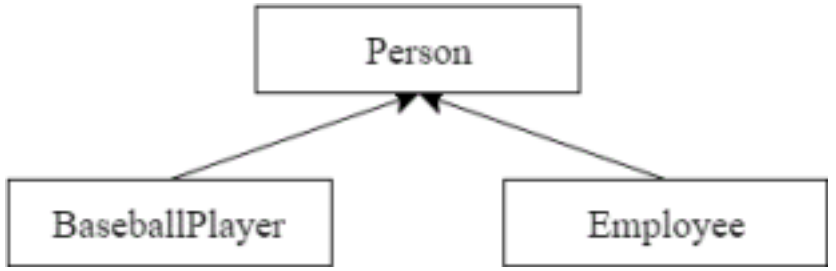
```
long m_employeeID;

  Employee(double hourlySalary = 0.0, int employeeID =
0): m_hourlySalary(hourlySalary),
m_employeeID(employeeID) {
 }

void printNameAndSalary() const
 {  //Note that Employee class can access its Base class public members directly.
    cout << m_name << ": " << m_hourlySalary << endl; }
 };
```

Creating Derived class Employee object

- Employee objects will have 4 member variables:
- ***m_hourlySalary*** and ***m_EmpID*** from Employee, and ***m_name*** and ***m_age*** from Person.
- The main() outputs name and salary of the employee object.
    - Employee inherits name from Person

```
int main() {
  // Create a new employee object
 Employee emp (1000, 25); //calls constructor with arguments hourlysalary, and empID
  // Assign it a name (we can do this directly because m_name is public) emp.m_name =
  "Smith";

emp.PrintNameandSalary();

    return 0;
}
```

Modified Base



class (Person)

- Now let's make the data members private and member functions public.
- And let's try to access m_name in the derived class Employee

```
#include <string>
class Person {
Private:
    string m_name;
    int m_age;
Public:
Person(string name = "", int age =
0): m_name( name ), m_age( age )
    {
    }
string getName() { return m_name; }
 int getAge() { return m_age; } };
```

Inheriting from the base class

• We publicly inherit the Employee class from the Person class

```
// Employee publicly inherits from Person
class Employee: public Person {
public:
    double m_hourlySalary;
    long m_employeeID;

    Employee(double hourlySalary = 0.0, int employeeID = 0) :
m_hourlySalary(hourlySalary), m_employeeID(employeeID) {
}
```

```
void printNameAndSalary() const ;

};
```


Accessing member of base class

It will generate an ERROR as m_name is a private member of base class and cannot be accessed directly in derived class.

```
void printNameAndSalary() {
```

//Note that Employee class CANNOT access its Base class private members directly.

```
        cout << m_name << ": " << m_hourlySalary << endl;
    }
```


Accessing member of base class

However, derived class can access private members of base class via

pubic member functions.

```
void printNameAndSalary()
{ //Note that Employee class CANNOT access its Base class
private members directly.
        cout << getname()<< ": " << m_hourlySalary << endl;
    }
```


Driver program to test Employee class

```
int main() {
    Employee emp;
    emp.m_name = "Ali"; //ERROR: m_name is private. // Print out the name
cout << emp.getName() <<endl;


    return 0;
}
```

Class exercise

- There is a student base class. It has the following :
  - Attributes: Name, age, RollNumber, CGPA
  - Behavior: ShowCGPA()
- Following two classes are derived from the student class.
  - UGStudent

- Attributes: string FYPtitle, string supervisorname
- Methods: showFYPtitle()

- PGStudent
  - Attributes: string Thesistitle, string supervisorname
  - Methods: showthesistitle()

'Protected' Access Specifier

- We have already covered the public and private access specifiers.
- There is a third access specifier 'Protected' which is only useful in the context of inheritance.
- A member declared as protected is accessible by the member functions within its class and any class immediately derived from it.
- It cannot be accessed by the function outside these two classes.

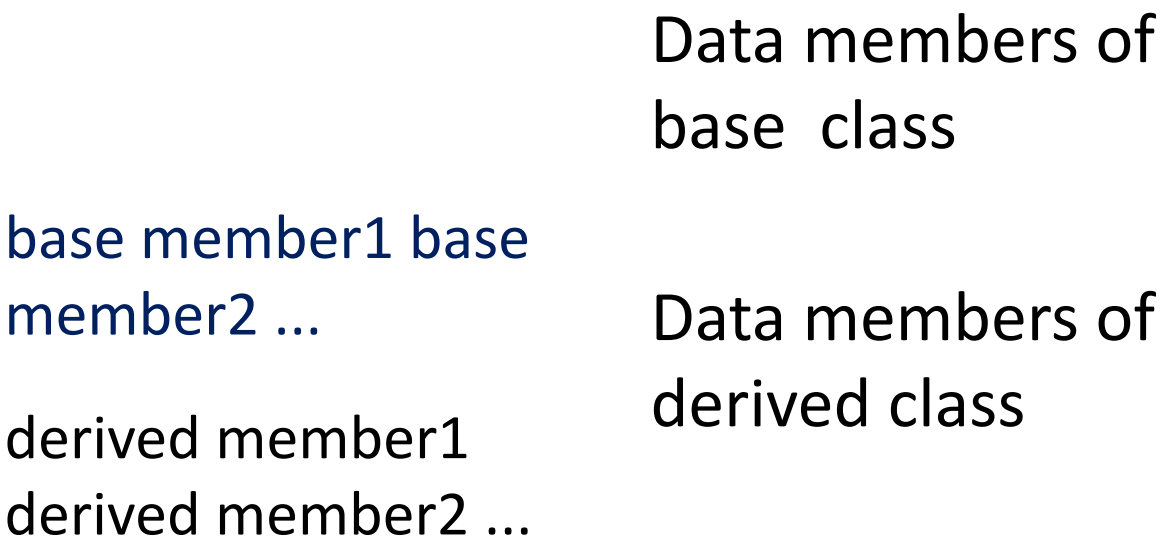
'Protected' Access Specifier

```
class Base {
private:
```

```
 int MyPrivateInt;
protected:
 int MyProtectedInt;
public:
 int MyPublicInt;
};
class Derived : Base {
public:
int foo1() { return MyPrivateInt;} // Won't compile!
int foo2() { return MyProtectedInt;} // OK. Does not need base class member function.
int foo3() { return MyPublicInt;} // OK
};

int main() {
 Base base;
 base.MyPublicInt = 1; // allowed: can access public members from outside class
base.MyProtectedInt = 2; // not allowed: can not access protected members from outside class
base.MyPrivateInt = 3; // not allowed: can not access private members from outside class

return 0;
}
```

Memory footprint

- Every object of derived class has an anonymous object of base class
- The object of derived class is represented in memory as follows



Constructors

- The anonymous object of base class must be initialized using constructor of base class
- When a derived class object is created the constructor of base class is executed before the constructor of derived class

base member1 base member2 ...

derived member1 derived member2 ...

Base class constructor

initializes the anonymous object

Derived class constructor initializes the derived class object

Constructors in Inheritance

- Constructors :
  - Derived class constructor implementation should call parent class constructor
  - Make call explicit in initialization list
  - Calling order: bottom-up
  - Execution order: top-down

Inheritance With Default Constructors

- Both Parent and child classes have default (no-argument constructors)
- In this case, the subclass constructor automatically and implicitly calls the Base class default constructor.
- When a derived class object is instantiated, constructor of the base class is called, initializing the base class private members.
- After the base class constructor completes execution, it returns control to derived class

constructor which initializes the derived class object.

```
class Person {
public:
        Person() {
cout << "Person Constructor...";
}
};
class employee : public Person {
public:
employee() {
cout << "Employee Constructor...";
}
};
int main() {
Employee
Emp; }
```

Output:

Person Constructor... Employee Constructor...

Calling base class Default Constructors explicitly

- It is also possible that the Employee class constructor explicitly calls the Person class constructor.
- For this purpose constructor initializer list can be used to explicitly call the base class constructor.

```
class Person {
public:
Person() {
cout << "Person Constructor...";
}
};
class employee : public Person {
public:
employee():Person() {
cout << "Employee Constructor...";
}
};
```

```
int main() {
Employee Emp;
}
```

Output:

Person Constructor... Employee Constructor...

Inheritance with non-default Constructor

- If default constructor of base class does not exist, then the compiler will try to generate a default constructor for base class and execute it before executing constructor of derived class
- If the user has given only an parameterized constructor for base class, the compiler will not generate default constructor for base class

• In Example below, Person constructor takes argument. But Employee constructor called is assuming the call to the default Person constructor without any arguments.

```
class Person {
Private:
…
Public:
Person(string name, int age):m_name( name ), m_age( age ) {
}
};
class Employee: Public Person {
Private:
…
Public:
Employee()//ERRO
      R { }
};

        int main() {
```

```
Employee Emp;
//ERROR }
```

Inheritance with non-default Constructor

- In case of non-default base class constructor, the derived class constructor will explicitly call the base class constructor using the constructor initializer list.

```
class Person {
Private:
    …
Public:
Person(string name, int age):m_name( name ), m_age( age )
    {
    }
 };
class Employee: Public Person {
Private:
    …
Public:
Employee(string a_name, int
  a_age):Person(a_name,a_age) { }
 };                                  int main() {
```

```
Employee Emp("Sue Smith",        25); }
```

Inheritance with non-default Constructor having default arguments

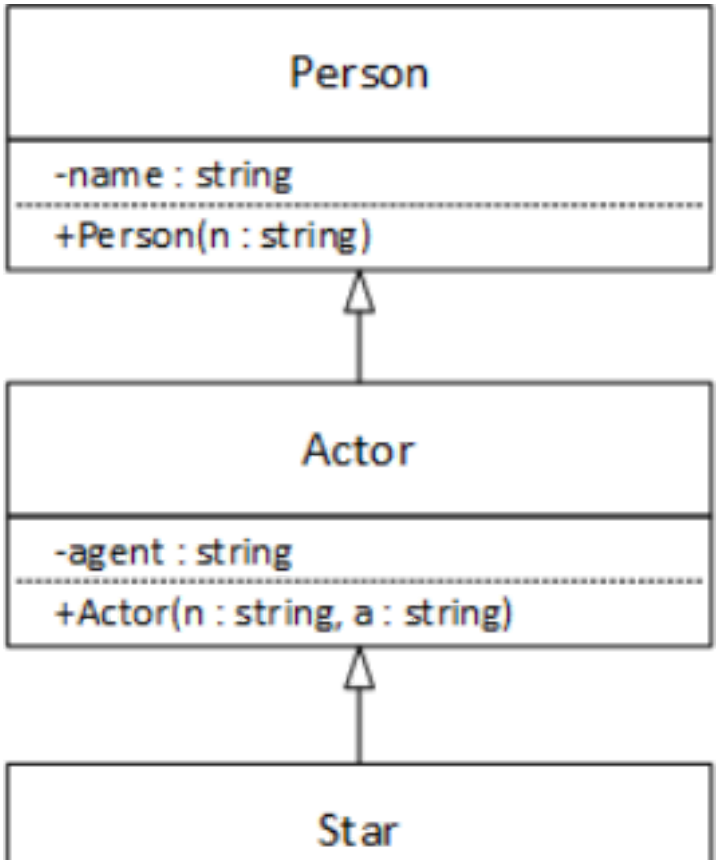
- In case of non-default base class constructor, the derived class constructor will explicitly call the base class constructor using the constructor initializer list.

```
class Person {
Private:
    …
Public:
Person(string name="", int age=0):m_name( name ), m_age( age )
    {
    }
 };
class Employee: Public Person {
Private:
    …
Public:
Employee(string a_name="", int
  a_age=0):Person(a_name,a_age) { }
 };
```

```
int main() {
Employee Emp1("Sue Smith", 25);
Employee Emp2;
}
```

Another Example:

```
class Person {
Private:
    …
Public:

 };
class Actor: Public Person {
Private:
    …
Public:

 };
```



```
Class Star: public Actor {
Private:
…
Public:
…
};
```

Another Example: Inheritance With Non-Default Constructors

• The child can only perform the initialization of direct base class through *base class initialization list* • The child can not perform the initialization of an indirect base class through *base class initialization list*

Another Example: Inheritance With Non-Default Constructors

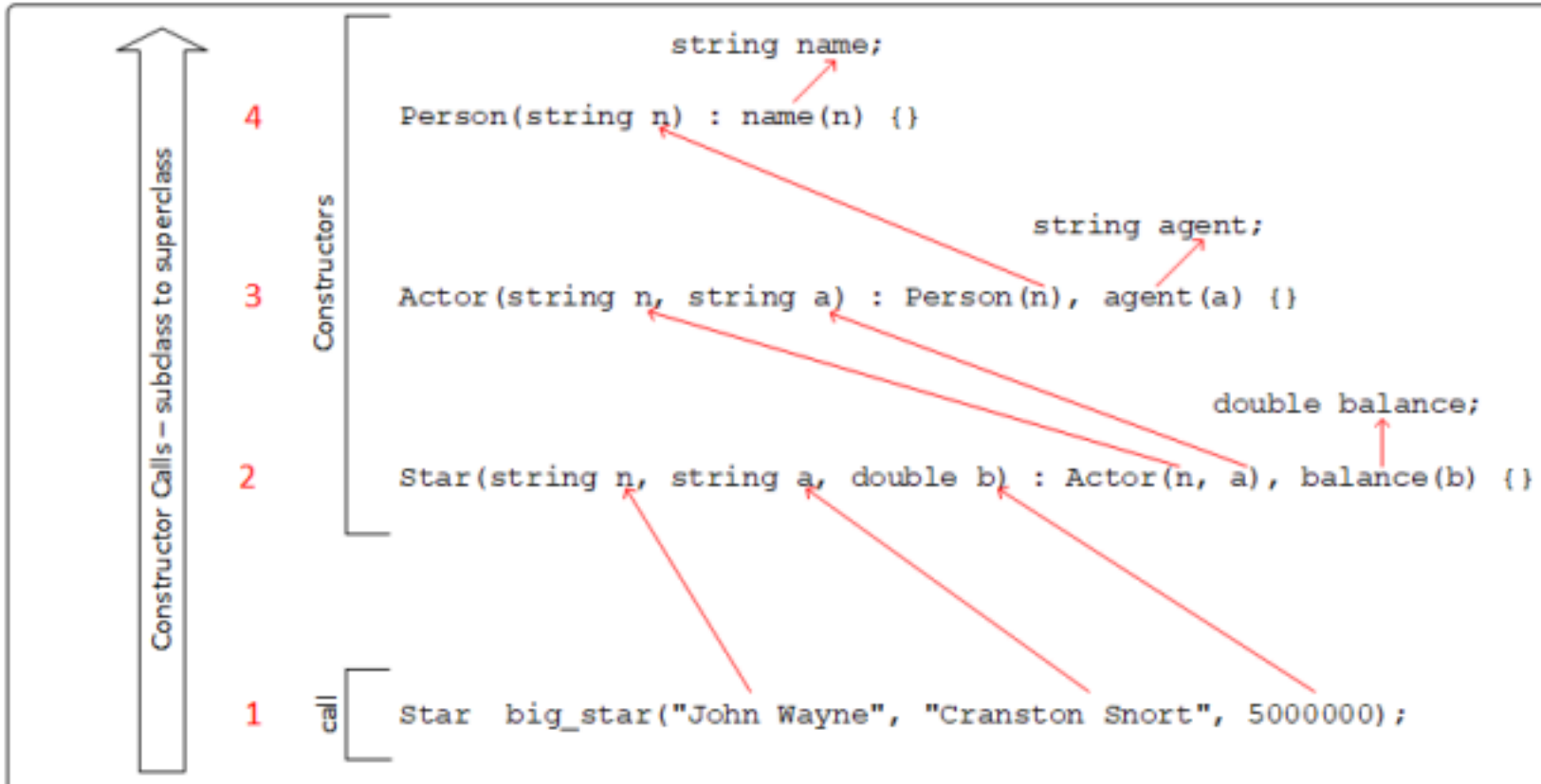


Figure 7. Data flow in chained constructor calls. The arrows indicate the direction that data, which originates in the function call at the bottom (1), flows upwards through the constructors, the initializer lists, and finally into the member variables.

1. The chain begins when the program instantiates an instance of `Star` (`big_star`) and passes three values into the constructor.
2. The `Star` constructor uses one parameter, b, to initialize its member variable, balance, and passes the values in the remaining tw parameters, n and a, to the `Actor` constructor
3. The `Actor` constructor uses one parameter, a, to initialize its member variable, agent, and passed the value in the remaining parameter, n, to the Person constructor.
4. The `Person` constructor uses the parameter, n, to initialize its member variable, name.

Constructors and Destructors in Base and Derived Classes

- Derived classes can have their own constructors and destructors
- 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

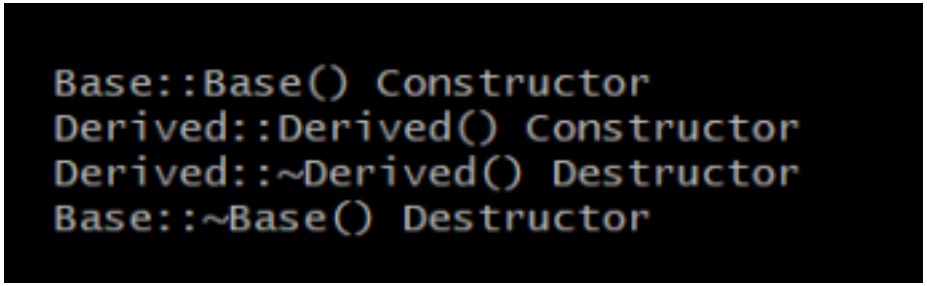
Destructors in Inheritance

- Destructors:
  - No explicit call to base class destructor

- Called automatically as object departs scope
- Calling order: bottom-up
- Execution order: bottom-up

Code example: Constructors and destructors in Inheritance

```
class Base {
 public: Base () {
 std::cout << "Base::Base()\n";
 }
 ~Base () {
 std::cout << "Base::~Base()\n";
 }
};
class Derived : public Base {
 public:
 Derived () : Base() {
 std::cout << "Derived::Derived()\n";
 }
 ~Derived () {
 std::cout << "Derived::~Derived()\n";
 }
};

int main() {
Derived D;
}
```



Class exercise

- There is a student base class. It has the following :
  - Attributes: Name, age, RollNumber, CGPA
  - Behavior: ShowCGPA()
- Following two classes are derived from the student class.
  - UGStudent
    - Attributes: string FYPtitle, string FYPsupervisorname
    - Methods: showFYPtitle()

- PGStudent
  - Attributes: string Thesistitle, string Thesissupervisorname
  - Methods: showthesistitle()

Dynamic Allocation in Base and Derived class

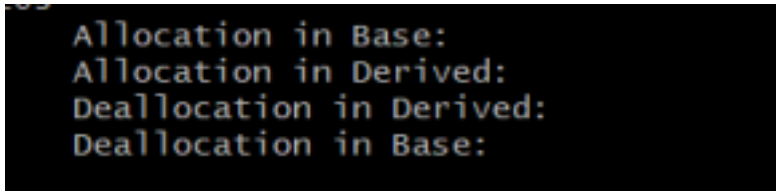
```
class Base {
 public:
 Base() { x = new int; }
~Base() { delete x; }
private:
 int* x;
};

class Der1 : public Base
 { public:
 Der1() { y = new int; }
```

- Destructors are explicitly called for dynamically

```
~Der1() { delete y; }
private:
 int* y;
};
void main() {
Der1* D = new Der1;
 delete D;
}
```

allocated objects using

delete:


Allocation in Base:
Allocation in Derived:
Deallocation in Derived:
Deallocation in Base: