Object-oriented programming

Lecture #8: Inheritance in C++

Outline

- Base and Derived Classes
  - Single Inheritance
  - Declaration of Derived Classes
  - Order of Constructor and Destructor Execution
  - Inherited Member Accessibility
- Multiple Inheritance
- Virtual Base Classes

Base and Derived Classes

- A **base class** is a previously defined class that is used to define new classes (now)
- A derived class inherits all the data and function members of a base class (in addition to its explicitly declared members)

Single Inheritance

- Implement an “is-a” relationship
- The derived class only has one base class

Declaring Derived Classes

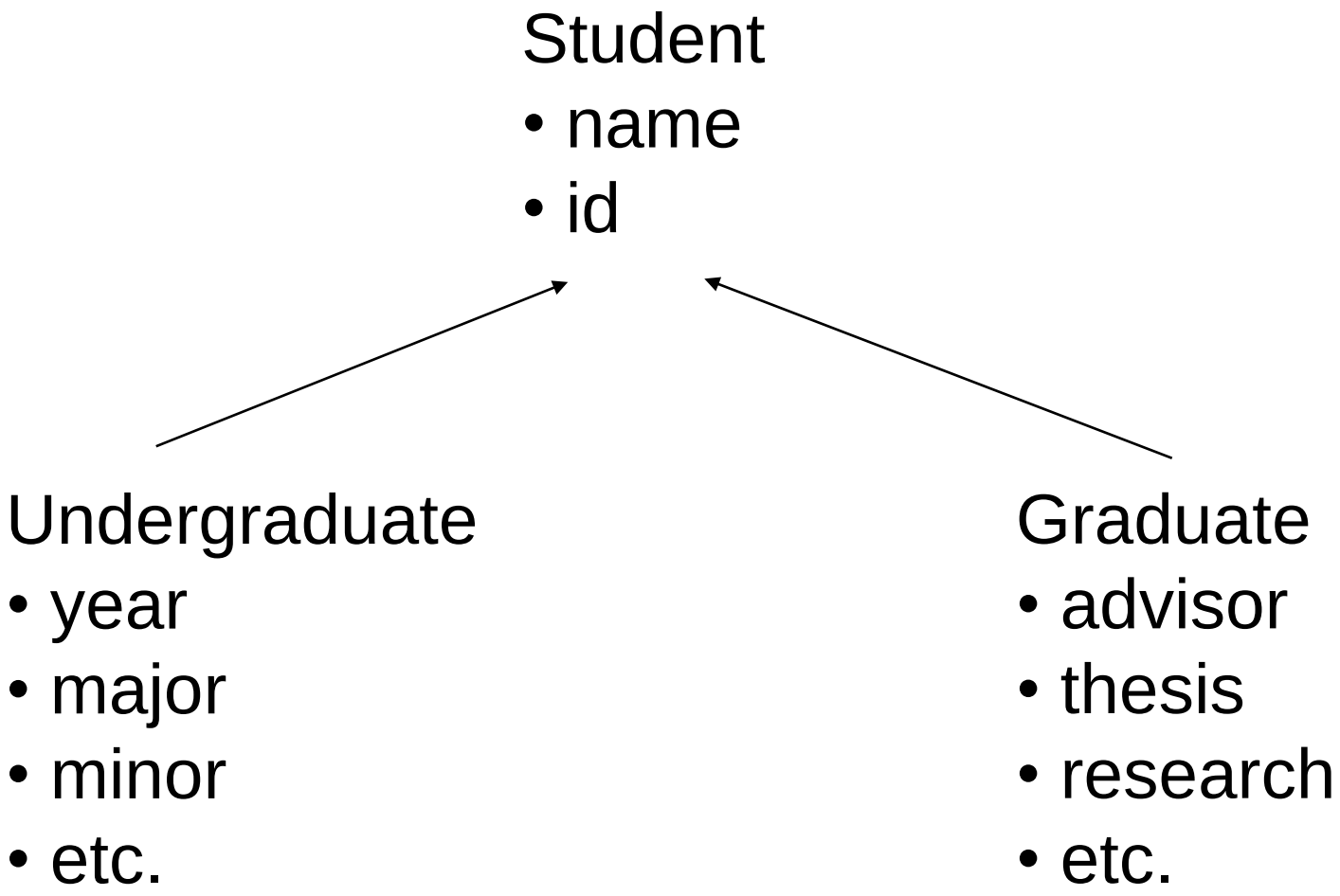
class class_name : access_specifier$_{opt}$ base_class {

    Member_list

};

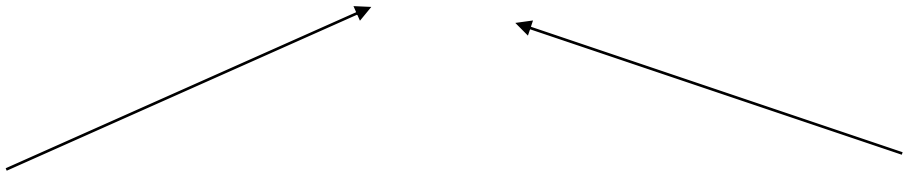
access_specifier ::= public|protected|private (default)

Example 1


Student
• name
• id
Undergraduate
• year
• major
• minor
• etc.
Graduate
• advisor
• thesis
• research
• etc.

Example 2

Publication:
- publisher
- date



Magazine:
- circulation
- # of issues per year

Book:
- ISBN
- author

Example 2

// Lec8_ex2-Publication.cpp

Different Views of an Employee

- Full-time or part-time
- Permanent or Temporary

(note different behavior, e.g. Social Insurance)

- How to define its base class?
- How to define derived classes based on this base class

Order of Constructor and Destructor Execution

- Base class constructors are always executed first
- Destructors are executed in exactly **the reverse order** of constructors

Example 3

// Lec8_ex2-Employee.cpp

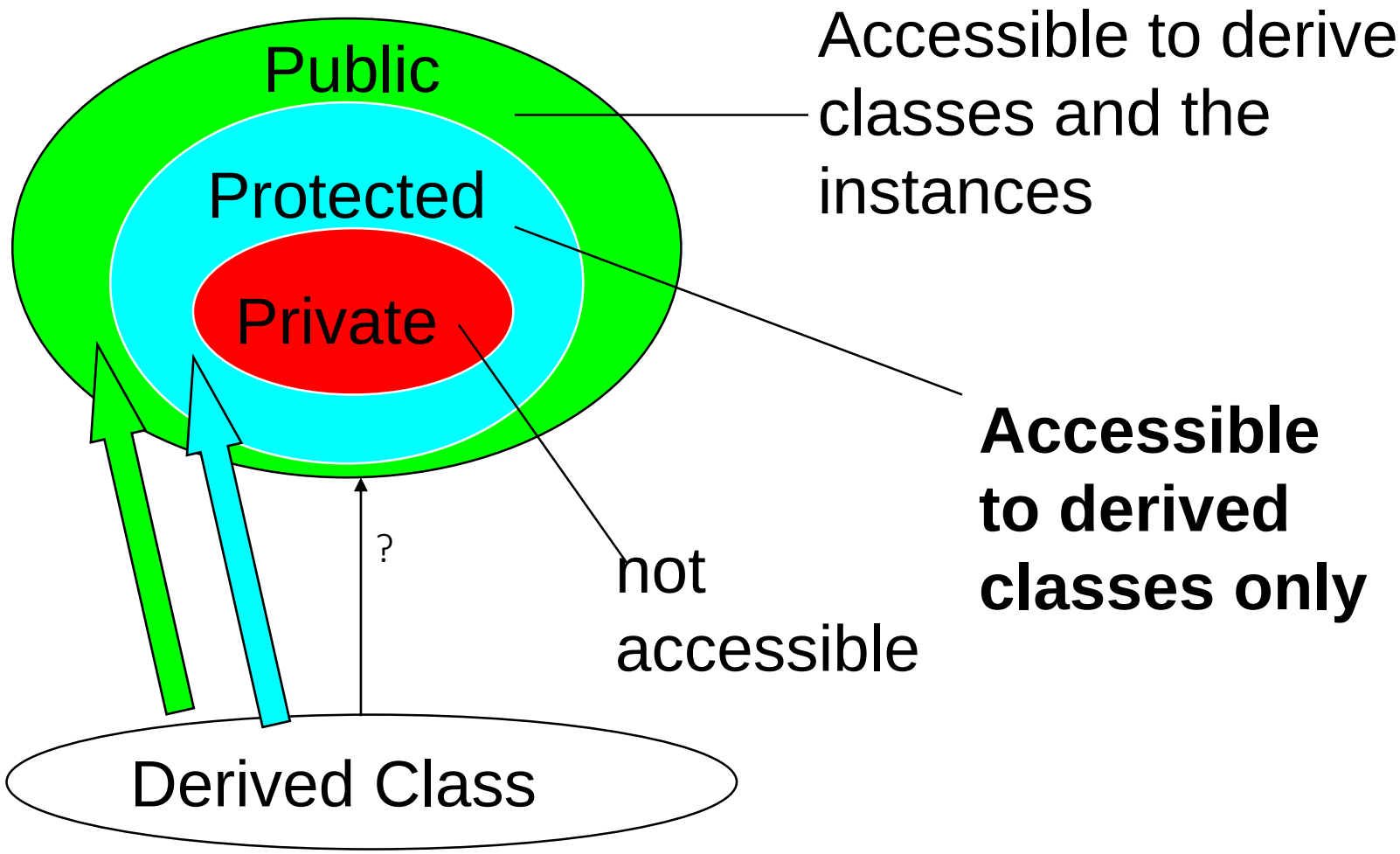
Overriding

- A function in the derived class with the same function name will override the function in the base class
- We can still retrieve the overridden functions by using the scope resolution operator “::”

Types of Class Members

- private
- protected
- public

Types of Class Members



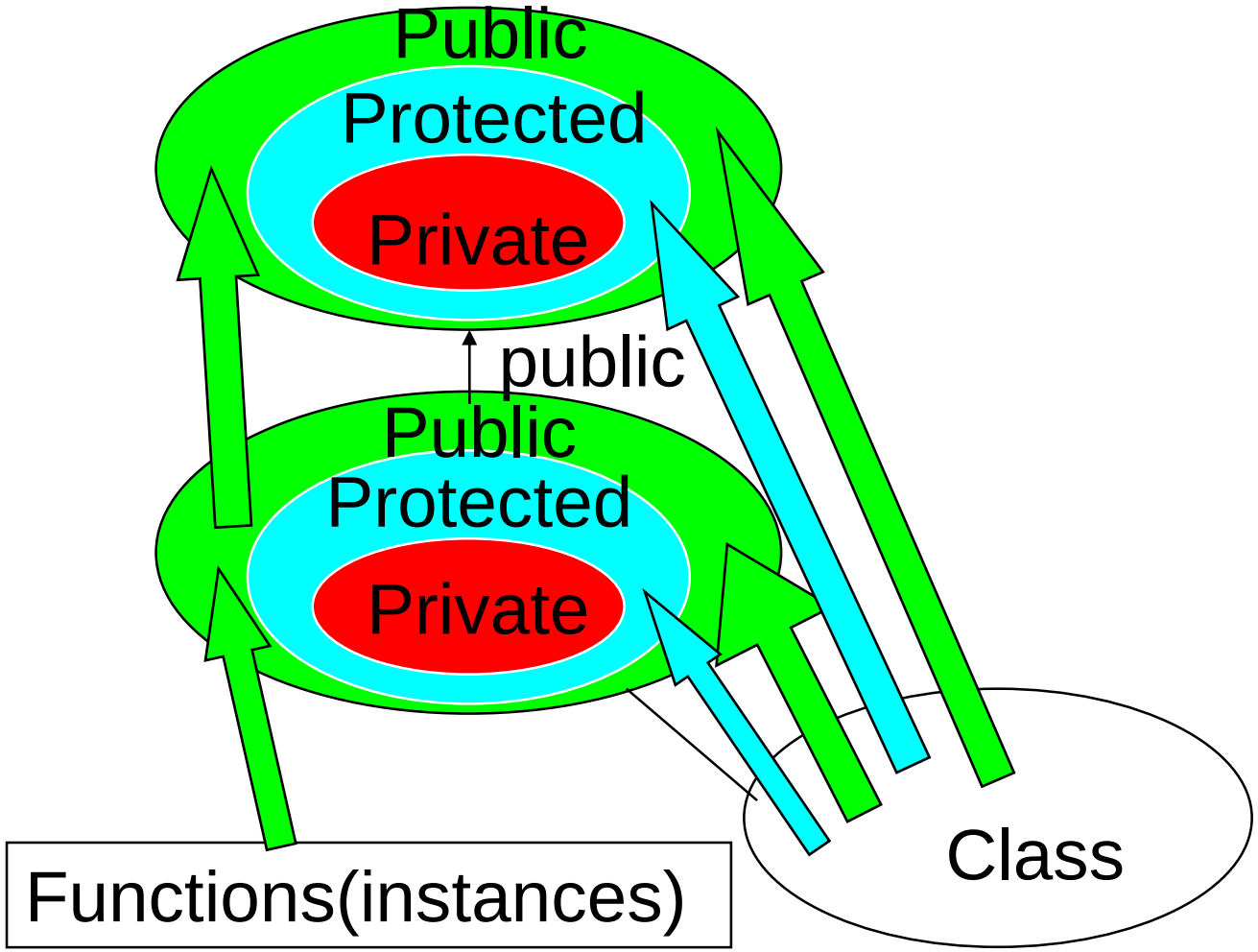
Types of Inheritance

- public
- private
- protected

public Inheritance

- public and protected members of the base class become respectively public and protected members of the derived class

public Inheritance



Example: Lec8_ex3-public-inher.cpp

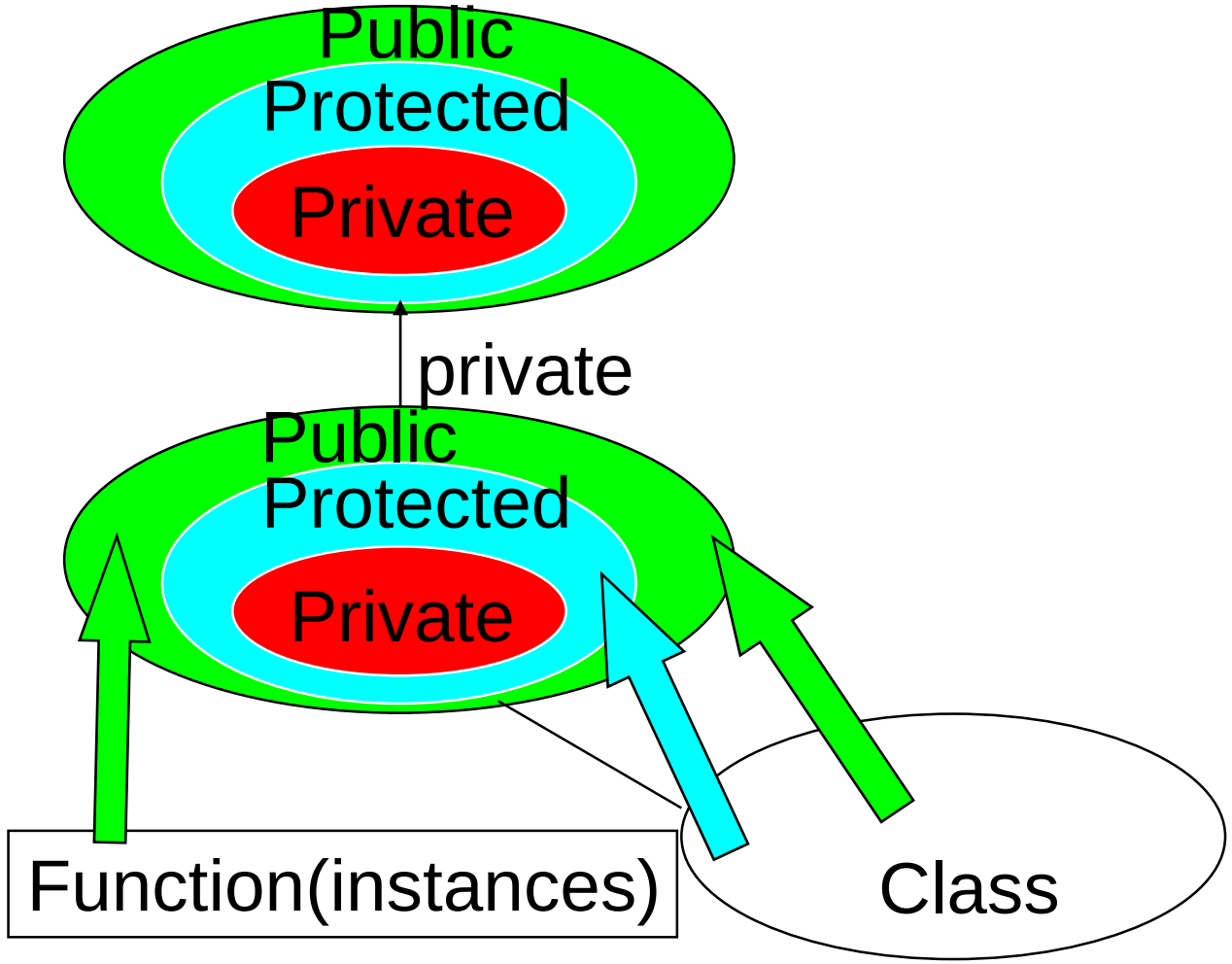
```
int main() {
    ..........
    p = aStack.removeFirst();
    ..............
return 0;
}
```

```
****** Creating an Item

Item::Item50
******* Creating a Stack

**** Stack::push() 50
Item::setPtr
List::putFirst() 50
**** Stack::pop()
List::removeFirst()
aStack.pop() 50

Item::setItem100
**** Stack::push() 100
Item::setPtr
List::putFirst() 100
Calling removeFirst() from aStack
List::removeFirst()
aStack.removeFirst() 100
```

private Inheritance

- public and protected members of the base class become private members of the derived class

private Inheritance



Example: Lec8_ex4-private-inher.cpp

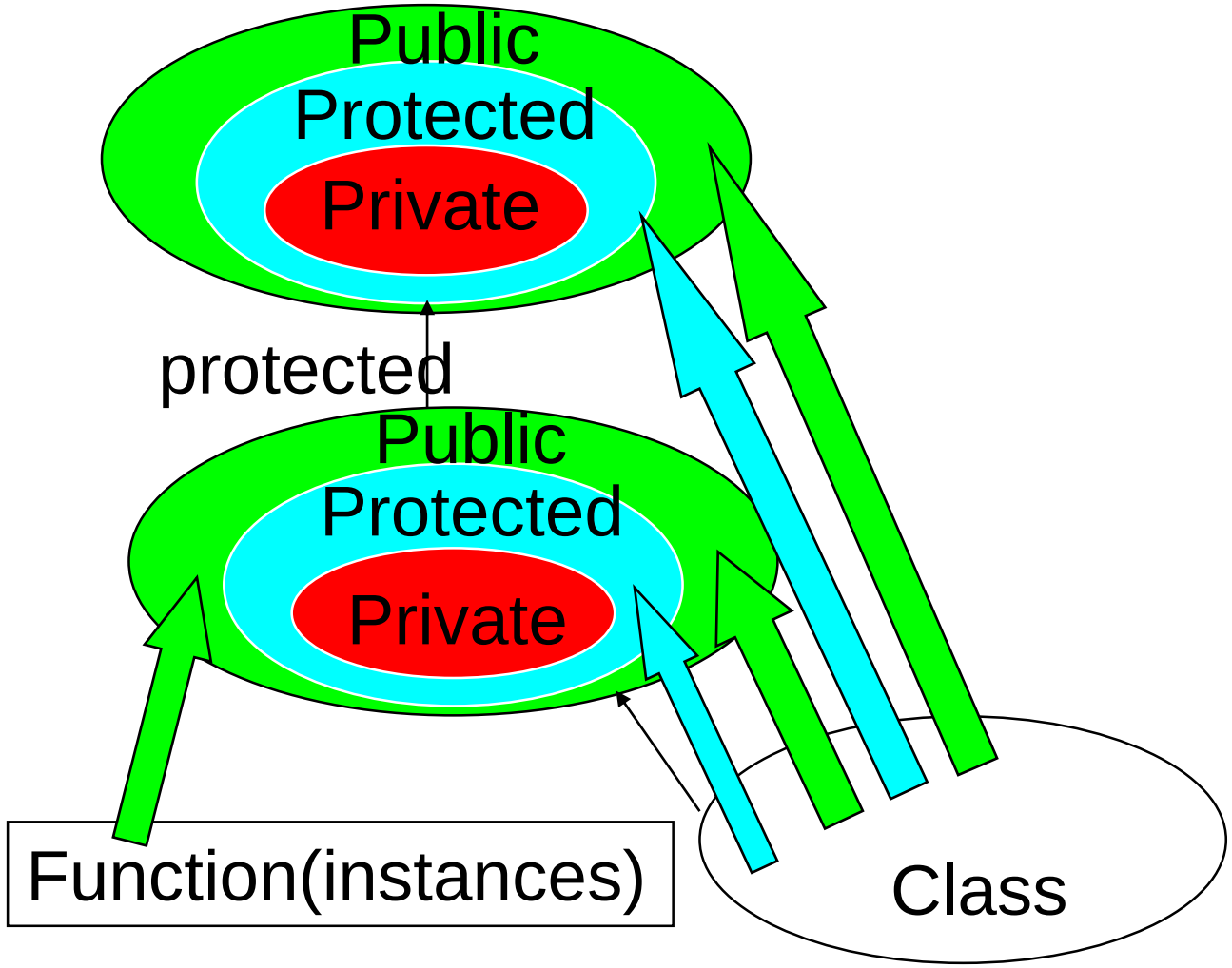
```
int main() {
    ……
    p = aStack.removeFirst(); //Error
    ……
return 0;
}
```

```
****** Creating an Item

Item::Item50
******* Creating a Stack

**** Stack::push() 50
Item::setPtr
List::putFirst() 50
**** Stack::pop()
List::removeFirst()
aStack.pop() 50

Item::setItem100
**** Stack::push() 100
Item::setPtr
List::putFirst() 100
Calling removeFirst() from aStack
List::removeFirst()
aStack.removeFirst() 100
```

protected Inheritance

- public and protected members of the base class become protected members of the derived class

protected Inheritance



Example: Lec8_ex5-protected-inher.cpp

```
int main() {
    ..........
    p = aStack1.removeFirst(); //Error
    ............
return 0;
}
```

```
Item::Item50
Item::setPtr
List::putFirst() 50
Stack1::push() 50
Stack1::pop()
List::removeFirst()
 aStack1.pop50

Item::setItem100
Item::setPtr
List::putFirst() 100
Stack1::push() 100
```

Constructors in Derived Classes

- When an object of a derived class is created, the constructor of the (derived) class must first call the constructor of the base class
- See Lec8_ex6-ctor_derived.cpp

Constructor- Initializers

```
class_name::class_name(param-list) : ctor-initializer {
// function body
}
```

- ctor-initializer is used to transfer the parameters to the constructors of the base-class
- See Lec8_ex7-ctor_init.cpp

Why using ctor-initializer?

- Without it, the default constructor for the base class would be called, which would then have to be followed by calls to access functions to set specific data members

Destructors

- Destructors are called implicitly starting with the last derived class and moving in the direction of the base class

Compatibility Between Base and Derived Classes

- An object of a derived class can be treated as an object of its base class
- The reverse is not true

Nested Class Scope

- A public or protected base class member that is hidden from the derived class can be accessed using the scope resolution operator “::”
- For example: ***base-class::member***
- On the contrary, the base class cannot access the members of its derived classes

Implicit Conversion of Derived Pointers to Base Pointers

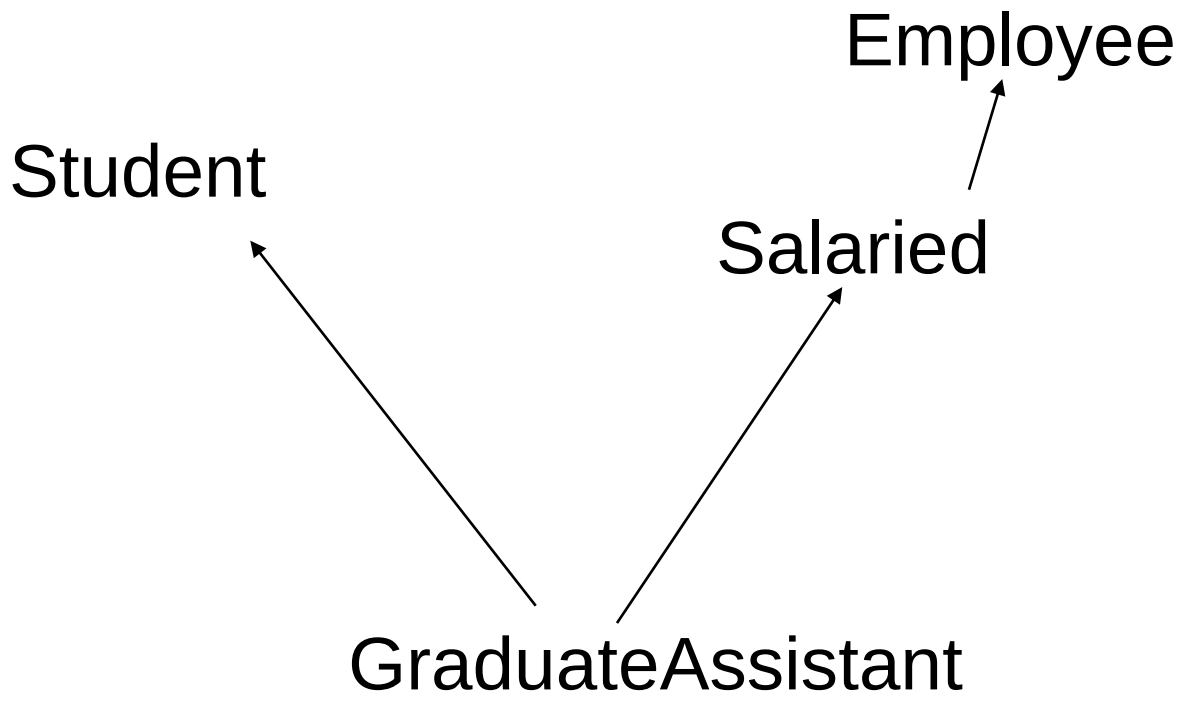
- A base type pointer can point to either a base object or a derived object

```
// Assume Point3D is derived from Point
Point3D * cp = new Point3D;
Point3D *cp1;
Point * p;
p = cp; //OK
cp1=p; //Error
cp1=(Point3D*) p;//OK
```

Casting Base Pointers to Derived Pointers

- A base pointer cannot be ***implicitly*** converted to a derived pointer
- This conversion is risky, because the derived is expected to contain more (attributes & behaviors) than the base object
- Forcing class users to use explicit casting often leads to poor code

Multiple Inheritance

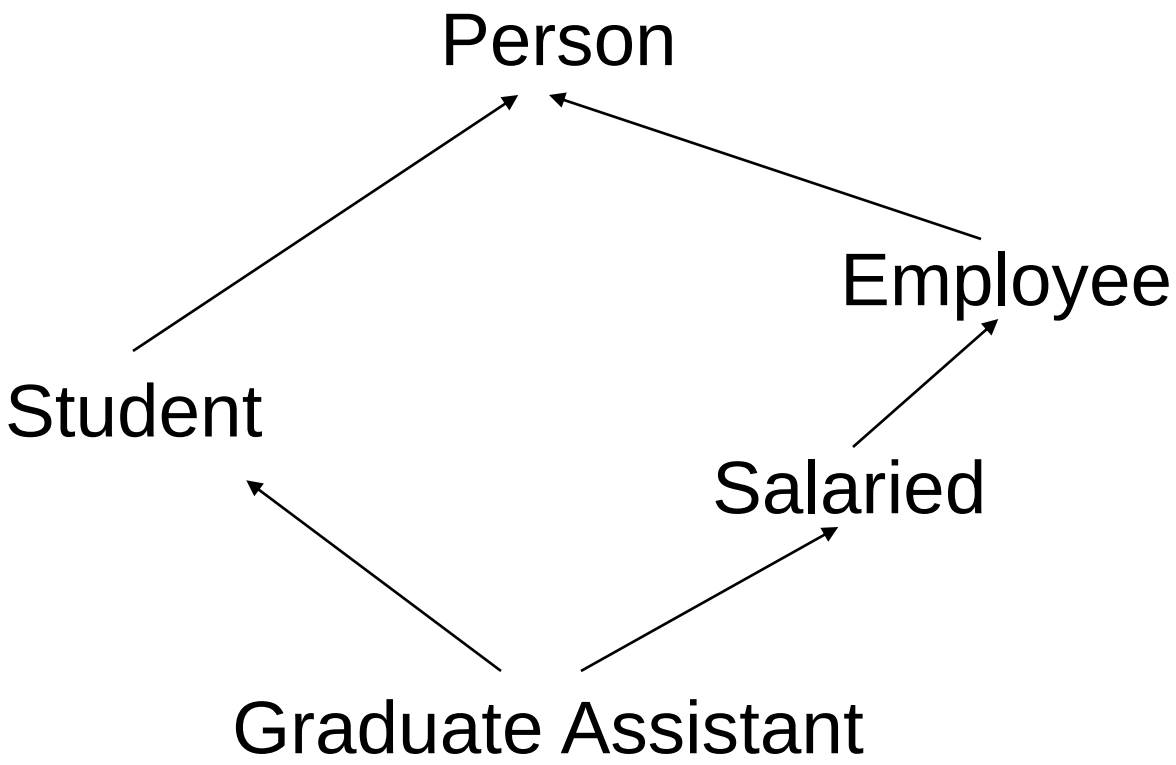


Example: Lec8_ex8_GradAssistant.cpp

Example

- Question: If we want to set a GradAssistant's age by calling setAge(), which setAge() should we use?
  - Student::setAge() or Salaried::setAge()
- Solution: Abstract (Virtual) base classes

Virtual Base Classes

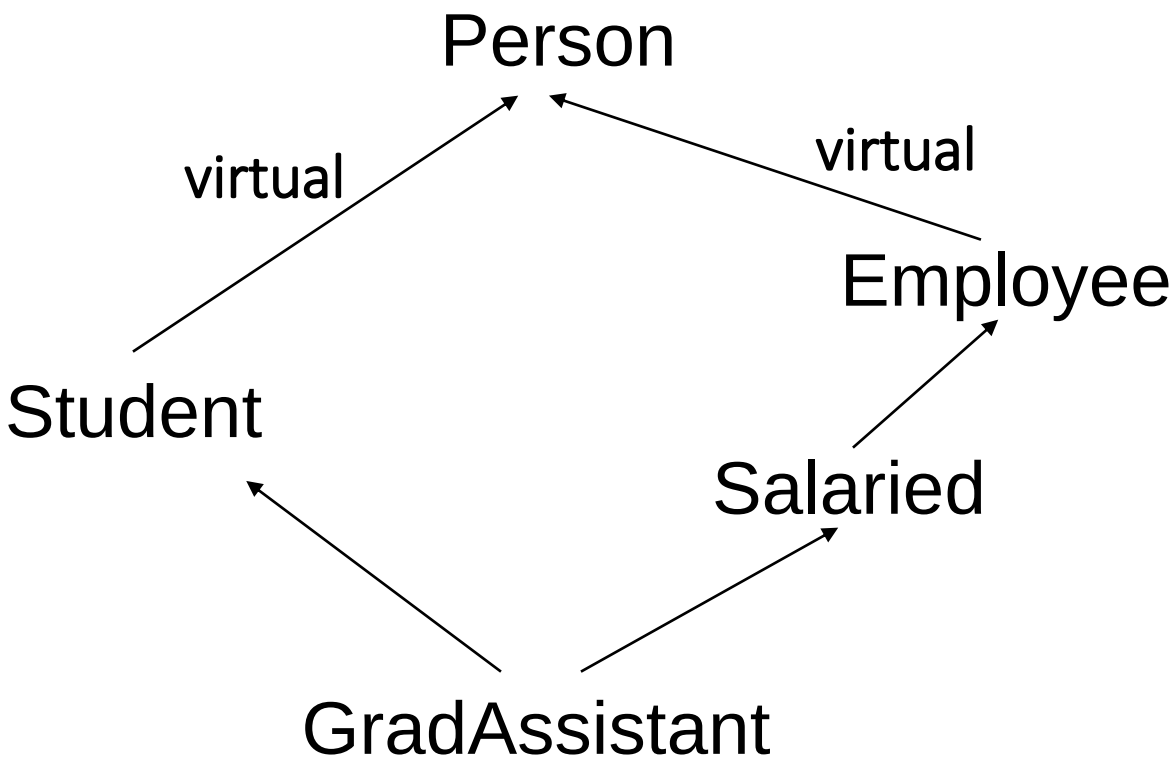


Example: Lec8_ex9_GradAssistant.cpp

Virtual Base Classes

- The function call to getAge() in GradAssistant::display() is ambiguous unless Person is inherited as a virtual base class
- Adding “virtual” lets the complier decide which function and which variable should be accessed

Virtual Base Classes



Virtual Base Classes

- When we use virtual inheritance, we are guaranteed to get only a single instance of the common base class. In other words, the **GradAssistant** class will have only a single instance of the **Person** class, shared by both the **Student** and **Employee** classes. By having a single instance of **Person**, we've resolved the compiler's immediate issue, the ambiguity, and the code will compile fine