Inheritance

Introduction

- Inheritance is one of the most powerful features of object-oriented programming.
- Inheritance allows creating new classes, called derived classes from existing base classes.
- Inheritance facilitates code reuse.
  - Functionality of the base class is inherited, and can be further refined in the derived class.
- Inheritance allows us to more faithfully model real-world scenarios.
  - E.g. B.Tech, Dual Degree, M.Tech are all students.

What is inherited?

- An object of the derived type contains all fields of the base type.
- An object of the derived type contains all methods of the base type.
- However, access permissions need to be respected.
- Constructors are not inherited.
  - Derived class needs its own constructor.

```
class Base{
  public:
      void fun() {
        cout << "in Base::fun" << endl;
      }
  protected:
      int n;
};
class Derived : public Base{
  public:
     void some() {
        n = 10;
        cout << "in Derived::some" << endl;
       }
};
int main(){
    Derived d;
    d.fun();
    d.some();
}
```

Access Permissions

- A derived class method inherits all public and protected fields and methods of the base class.
  - It can access all methods and fields of itself.
- A derived class method cannot access any private field or method of base class.

	public	protected	private
class	✓	✓	✓
children	✓	✓	✗
rest	✓	✗	✗

Constructors

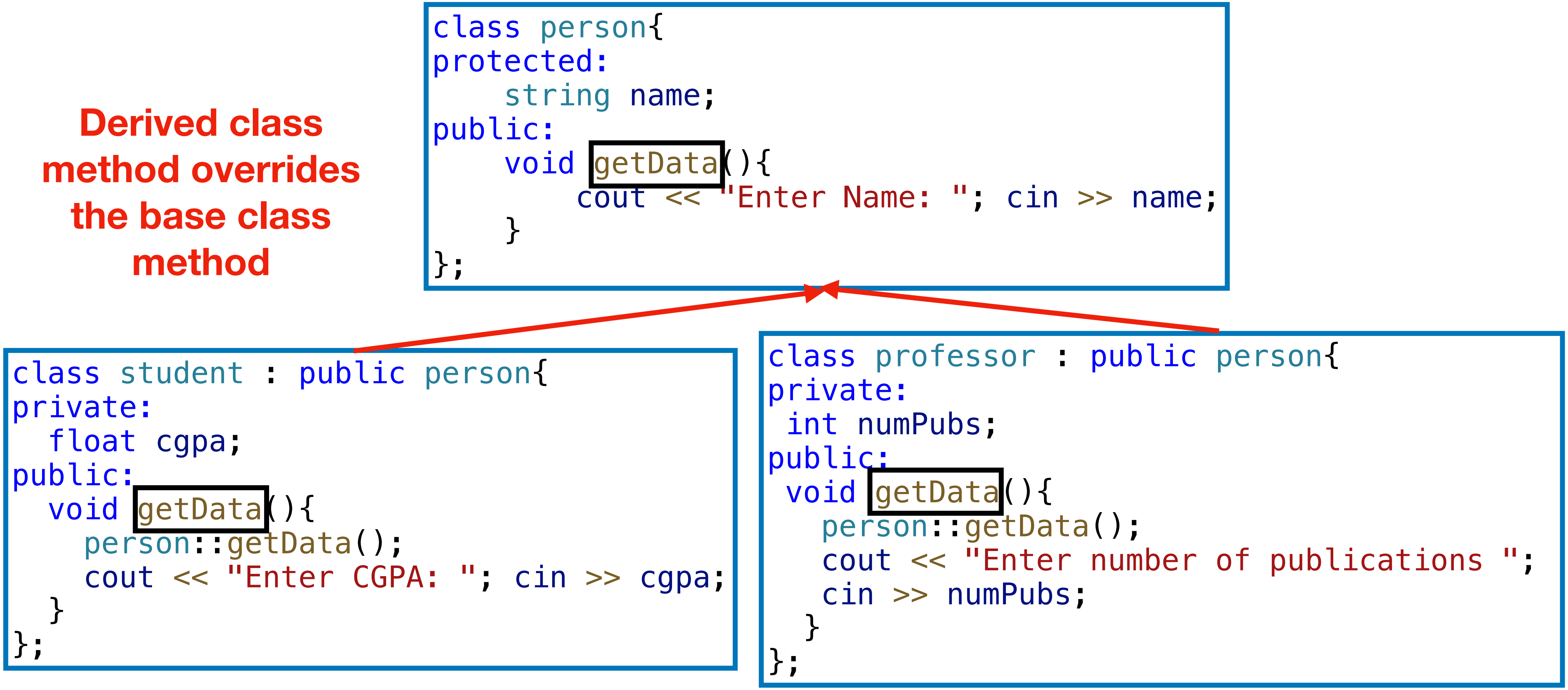
```
class IntCell{
public:
    explicit IntCell(int initialValue=0)
        : storedValue(initialValue) {}
    int read() const {return storedValue;}
    void write(int x) {storedValue = x;}
protected:
    int storedValue;
};

class Counter: public IntCell{
public:
    Counter(int initialValue=0): IntCell(initialValue) {}
    Counter operator ++() {return Counter(++storedValue);}
};
int main()
{
    Counter c;
    ++c;
    cout << c.read() << endl;
}
```

Derived class constructor calling the base class's constructor

Overriding

- A derived class can redefine a method from the base class.
- The derived class method hides the base class method.
  - A derived class object will call the derived class method, instead of the base class method.
  - This phenomenon is called overriding.

Overriding



Derived class method overrides the base class method

```
class person{
protected:
    string name;
public:
    void getData(){
        cout << "Enter Name: "; cin >> name;
    }
};
```

```
class student : public person{
private:
  float cgpa;
public:
  void getData(){
    person::getData();
    cout << "Enter CGPA: "; cin >> cgpa;
  }
};
```

```
class professor : public person{
private:
 int numPubs;
public:
 void getData(){
   person::getData();
   cout << "Enter number of publications ";
   cin >> numPubs;
  }
};
```

Pointers and Inheritance

- A base class pointer can point to a derived class object.
    - Can access public members of base class.
- This mechanism is extremely useful in uniformly handling all objects derived from the same base class.
    - We can call overridden methods of different derived classes using the same pointer of base class type.
    - To use this, the overridden method in the base class must be declared as virtual.

Example

```
class Base{
public:
    virtual void show()
    {cout << "in Base\n";}
};
class Derv1: public Base{
public:
    void show()
    {cout << "Derv1\n";}
};
class Derv2: public Base{
public:
    void show()
    {cout << "Derv2\n";}
};
```

Won’t work without ‘virtual’ keyword

```
int main(){
    Base * ptr;
    Derv1 d1;
    Derv2 d2;
    ptr = &d1;
    ptr->show();
    ptr = &d2;
    ptr->show();

}
```

Prints:
Derv1
Derv2

Example

```
class Base{
public:
    void show()
    {cout << "in Base\n";}
};
class Derv1: public Base{
public:
    void show()
    {cout << "Derv1\n";}
};
class Derv2: public Base{
public:
    void show()
    {cout << "Derv2\n";}
};
```

```
int main(){
    Base * ptr;
    Derv1 d1;
    Derv2 d2;
    ptr = &d1;
    ptr->show();
    ptr = &d2;
    ptr->show();

}
```

Prints:

in Base

in Base

Binding

```cpp
class Base{
public:
    virtual void show()
    {cout << "in Base\n";}
};
class Derv1: public Base{
public:
    void show()
    {cout << "Derv1\n";}
};
class Derv2: public Base{
public:
    void show()
    {cout << "Derv2\n";}
};
```

```cpp
int main(){
    Base * ptr;
    Derv1 d1;
    Derv2 d2;
    int input;
    cin >> input;
    if (input > 10)
        ptr = &d1;
    else
        ptr = &d2;
    ptr->show();
}
```

How does the compiler know which show method to call?

Dynamic Binding

- In general, the method invoked cannot be known at compile time.
    - The information required to invoke the appropriate method is only available at run-time.
- Compiler generates code to maintain a runtime table of pointer references, called virtual function table.
    - This phenomenon is called dynamic binding.
- In general, virtual functions require dynamic binding, while non-virtual functions use static binding.

```
int main(){
    Base * ptr;
    Derv1 d1;
    Derv2 d2;
    int input;
    cin >> input;
    if (input > 10)
        ptr = &d1;
    else
        ptr = &d2;
    ptr->show();
}
```

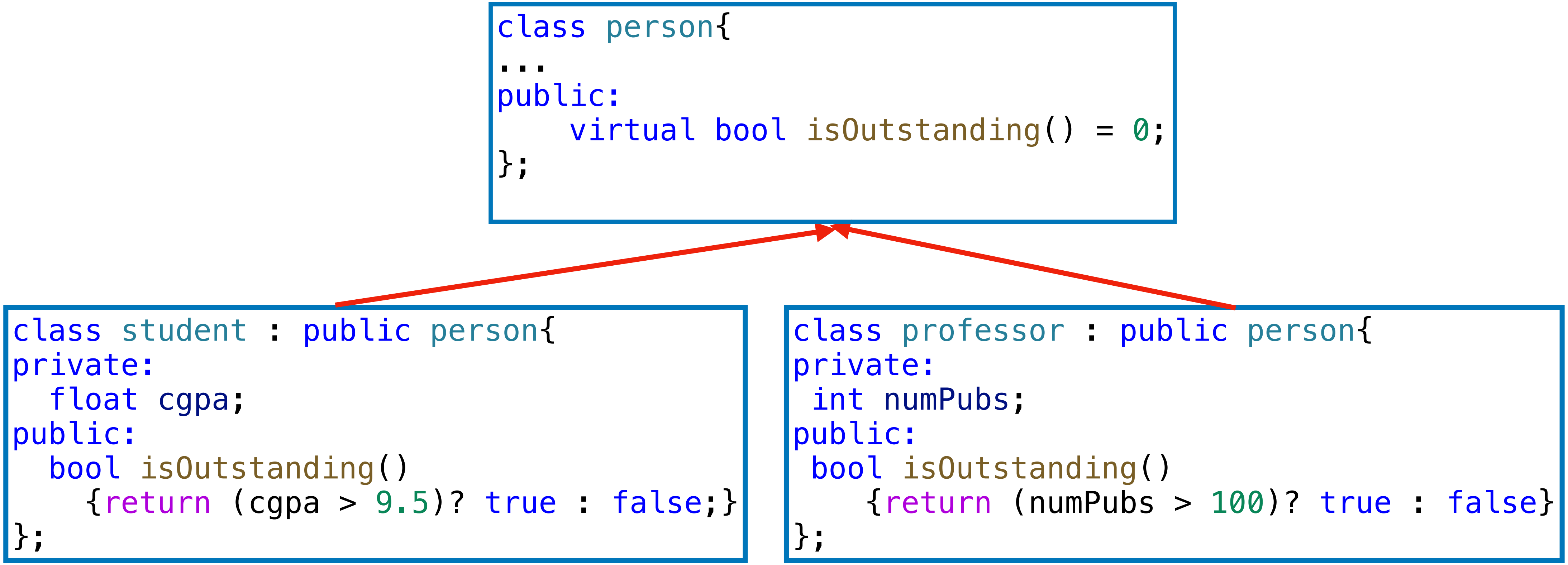
Abstract Classes and Pure Virtual Functions

- If a virtual function in a base class is always overridden in a derived class, it can be declared as pure.
    - The base class now becomes an abstract class.
- It is not possible to declare or instantiate an object of an abstract class.
- Every derived class of an abstract class must override all pure virtual functions.

Pure virtual function

```
class Base{
public:
    virtual void show() = 0;
};
class Derv1: public Base{
public:
    void show()
    {cout << "Derv1\n";}
};
class Derv2: public Base{
public:
    void show()
    {cout << "Derv2\n";}
};
```

Abstract Class: Example



```
class person{
...
public:
    virtual bool isOutstanding() = 0;
};
```

```
class student : public person{
private:
  float cgpa;
public:
  bool isOutstanding()
    {return (cgpa > 9.5)? true : false;}
};
```

```
class professor : public person{
private:
 int numPubs;
public:
 bool isOutstanding()
    {return (numPubs > 100)? true : false}
};
```

Overloading vs. Overriding

	Overloading	Overriding
Purpose	Readability	Change of functionality
Place	Within a class/Globally	Derived class
Parameters	Must be different	Must be same
Polymorphism	Compile time	Run time or Compile time

Overloading and Overriding do not mix

```
class Base{
public:
    virtual void show()
    {cout << "in base\n”;}
    virtual void show(int n)
    {cout << "in base " << n << endl;}

};
class Derv1: public Base{
public:
    void show()
    {cout << "Derv1\n";}
};
```

```
int main(){
    Derv1 d1;
    d1.show();
    d1.show(5);
}
```

Gives compilation error

show in derived class overrides all overloaded versions in base class

“Using” Base class members

```
class Base{
public:
    virtual void show()
    {cout << "in base\n”;}
    virtual void show(int n)
    {cout << "in base " << n << endl;}

};
class Derv1: public Base{
public:
    using Base::show;
    void show()
    {cout << "Derv1\n";}
};
```

```
int main(){
    Derv1 d1;
    d1.show();
    d1.show(5);
}
```

Prints:
Derv1
in base 5

Brings all overloaded versions in scope

Static members

- Each object has its own copy of class member fields.
- However, for static members, only one copy is created which is shared across all objects.
- Static members can even be accessed without creating any objects.
- Static functions can only used static fields.

Example

```
class foo
{
  private:
    static int count;
    int id;
  public:
    foo() {id = count++;}
    static int getCount() {return count;}
    int getID() {return id;}
};
```

```
int foo::count = 0;

int main()
{
  foo f1, f2, f3;
  cout << "No of objects " <<
       foo::getCount() << endl;
  cout << "f1's ID " << f1.getID() << endl;
  cout << "f2's ID " << f2.getID() << endl;
  cout << "f3's ID " << f3.getID() << endl;
}
```

Prints:

```
No of objects 3
f1's ID 0
f2's ID 1
f3's ID 2
```