Association, Aggregation, and Composition

(CS 217)

Dr. Naveed Ahmad,

Department of Computer Science,
National University of Computer & Emerging Sciences,
Islamabad Campus

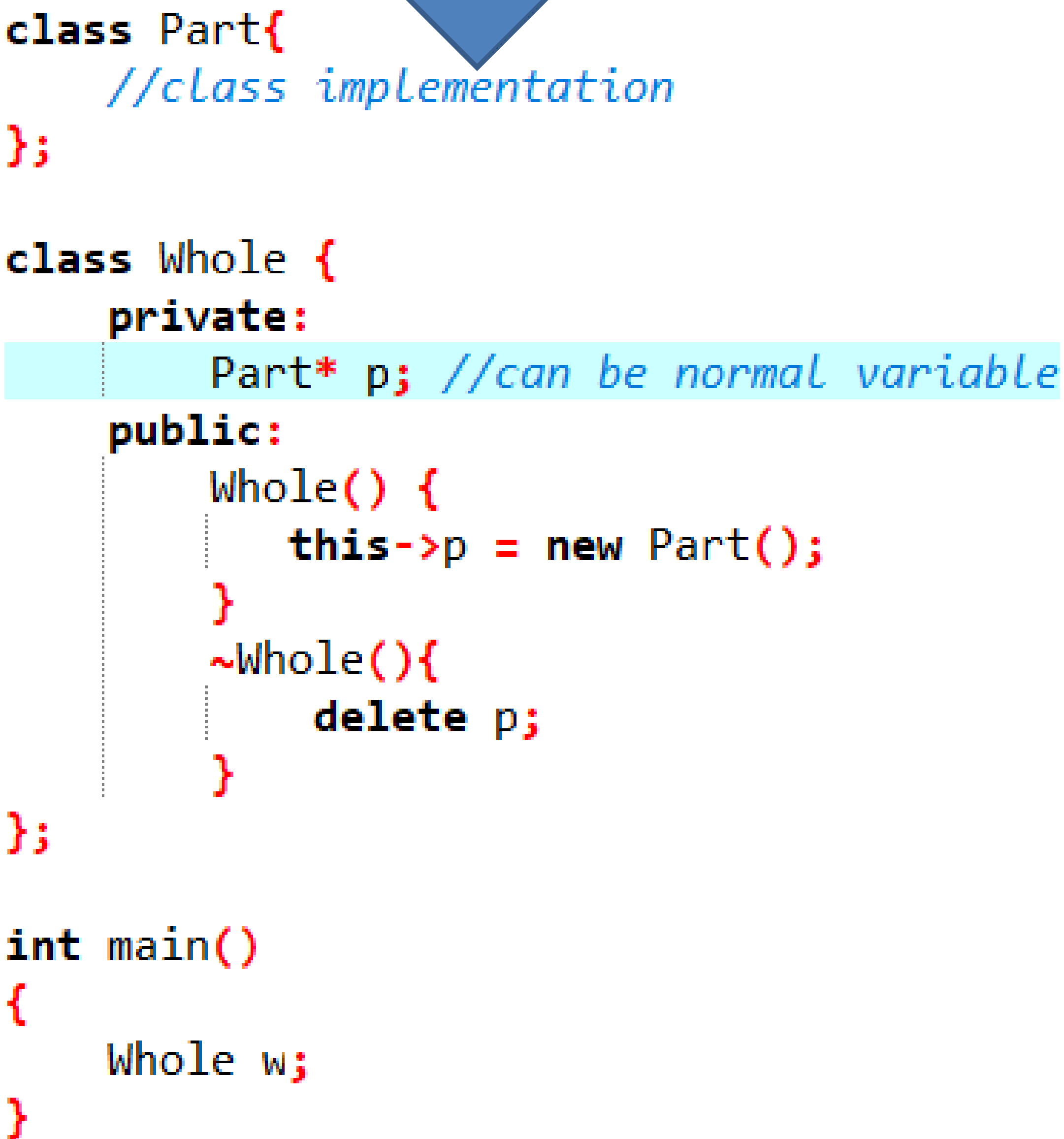
Implementing aggregation VS composition

- Aggregation
  - **Parts** are **added** as **references** or **pointers**
  - **Whole** is **not responsible** for creation and deletion
  - Whole takes the objects it is going to point to as: 1) **constructor parameters**; 2) **parts are added later via access functions**
  - Parts exists **outside the scope** of whole

- Composition
  - **Parts** are **added** as **normal variables** (or pointers)
  - **Whole** is **responsible** for **creation** and **deletion**

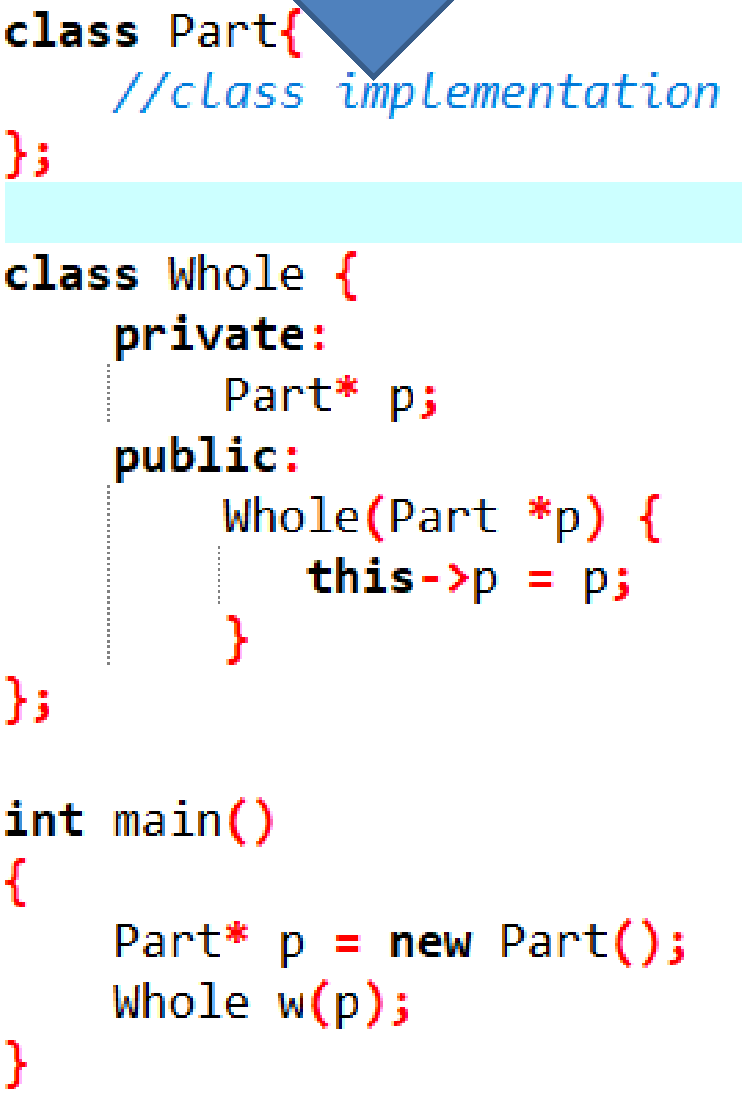
Examples

Composition



```cpp
class Part{
    //class implementation
};

class Whole {
    private:
        Part* p; //can be normal variable
    public:
        Whole() {
           this->p = new Part();
        }
        ~Whole(){
            delete p;
        }
};

int main()
{
    Whole w;
}
```

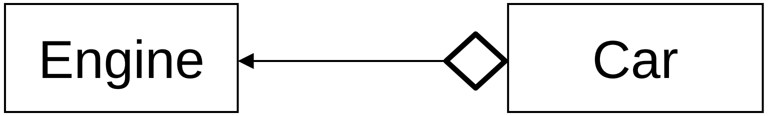
Aggregation



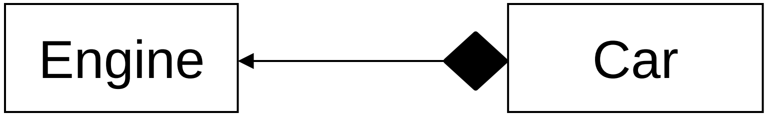
```cpp
class Part{
    //class implementation
};

class Whole {
    private:
        Part* p;
    public:
        Whole(Part *p) {
            this->p = p;
        }
};

int main()
{
    Part* p = new Part();
    Whole w(p);
}
```

Aggregation or Composition

- When to do what?



Repair shop software



Car performance software

Implement the simplest relationship that meets your needs!!!

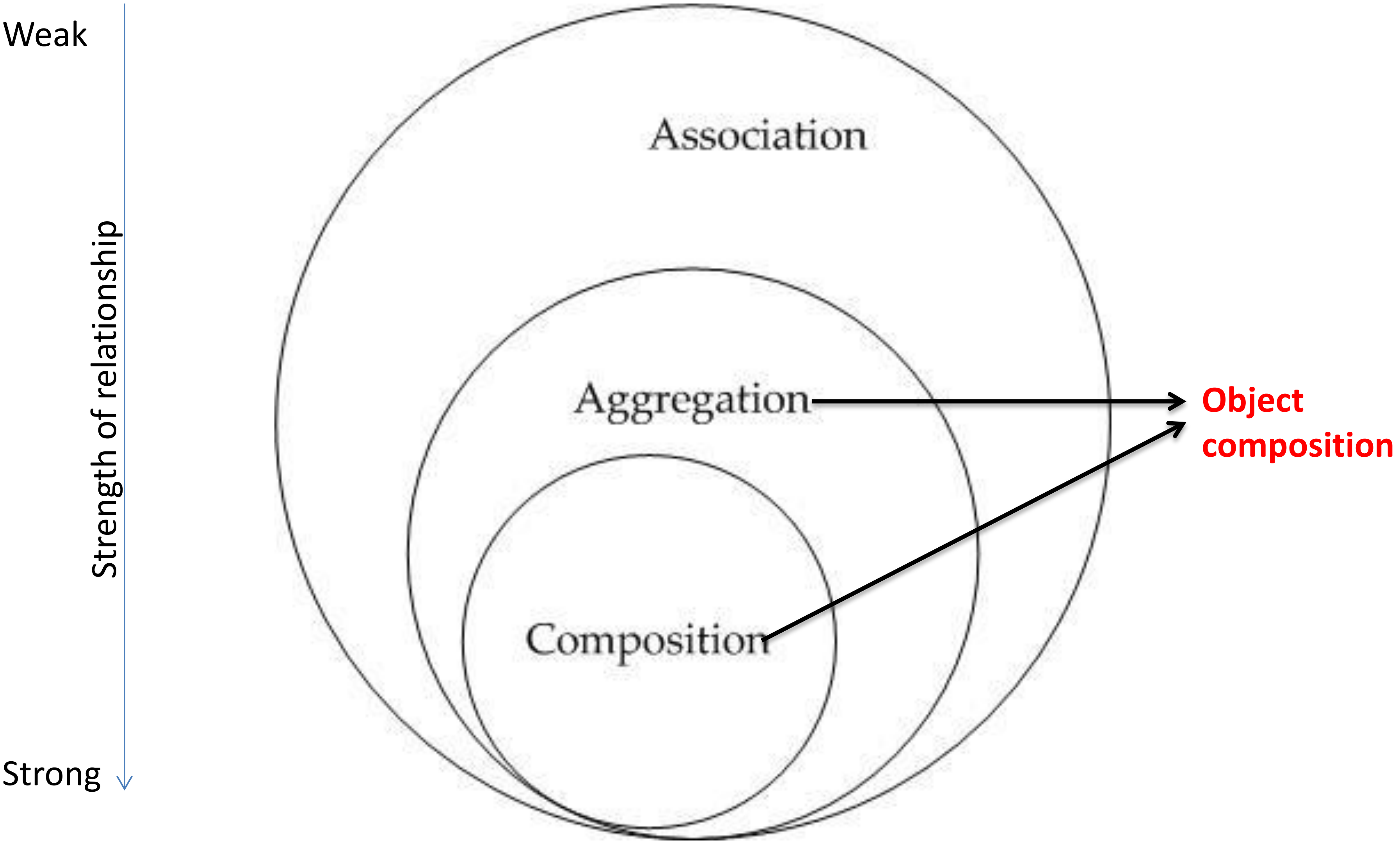
Not the one that seems like it would fit best in a real-life context.

Aggregation/Composition - recap

- Object composition
  - Composition
  - Aggregation
- Used to model relationships where a whole is built from one or more parts

Part 4
Association

Relationships between Objects



Weak

Strength of relationship

Strong

Association

Aggregation

Composition

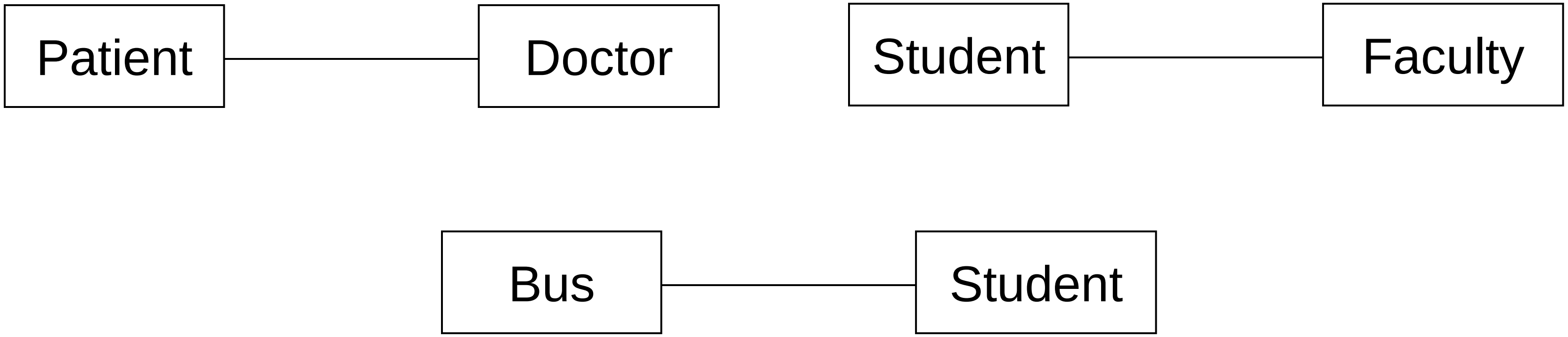
Object composition

Association

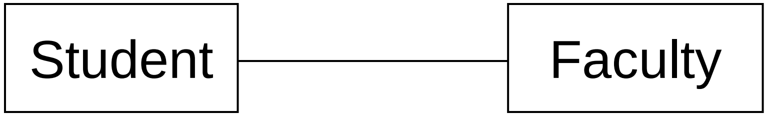
- A **weaker** type of relationship
- Two otherwise **unrelated objects**
- There is **no** implied **whole/part relationship**
- Models a **uses-a** relationship

Association

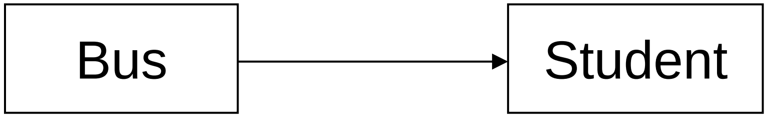
Composition	Aggregation	Association
Whole/Part relationship	Whole/Part relationship	**Associated object is unrelated**
Associated object can belong to only **one object**	Associated object can belong to multiple objects	Associated object can belong to multiple objects
Unidirectional	Unidirectional	**Bidirectional**



Association



- The teacher clearly has a relationship with his students and *vice versa*
- It’s not a part/whole (object composition) relationship
- A teacher can see many students
- A student can see many teachers
- Neither of the object’s lifespans are tied to the other.
- Bidirectional



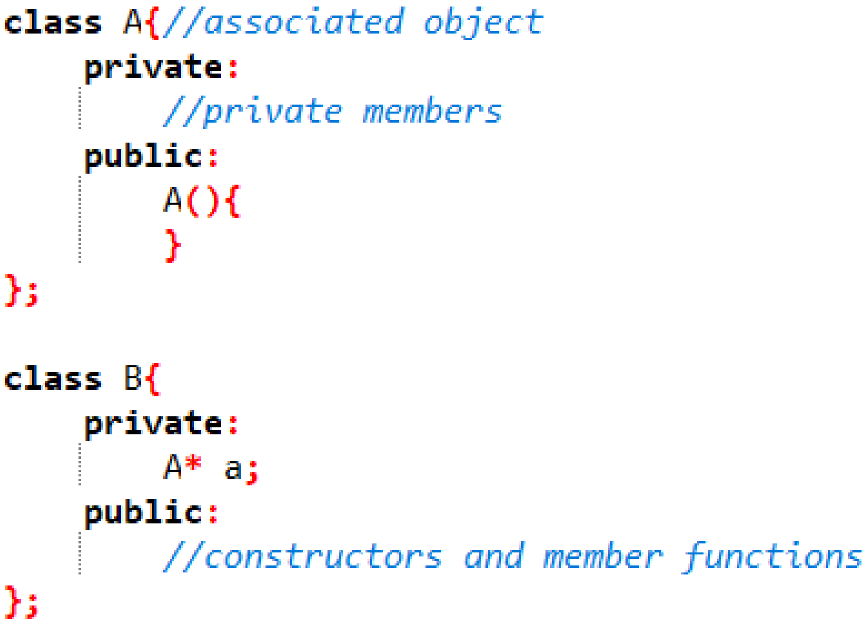
- A student has a relationship with the route bus
- Its not a part/whole relationship
- Multiple students can be on a certain route
- Neither of the object’s lifespans are tied to the other
- Unidirectional

Association tests

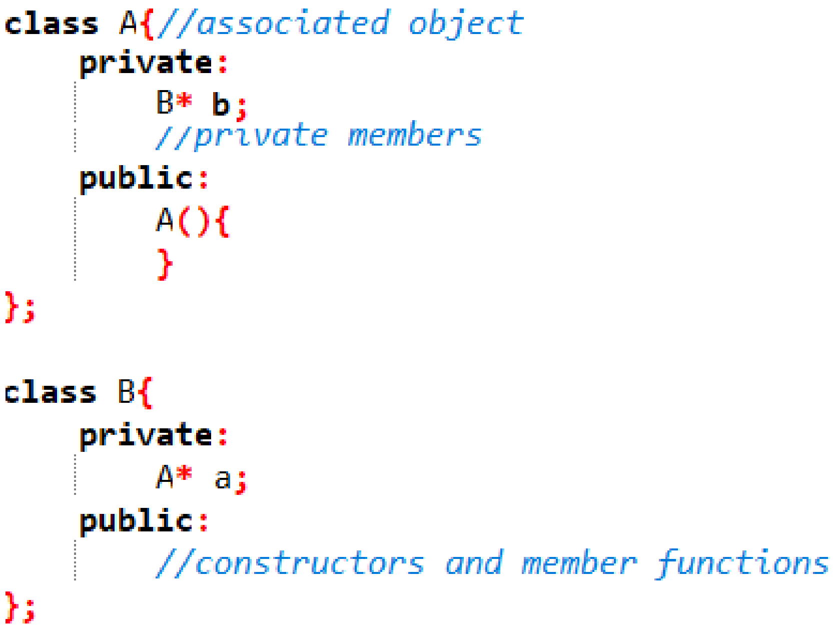
- The associated object (member) is otherwise **unrelated** to the object (class)
- The associated object (member) can belong to **more than one object** (class) at a time
- The associated object (member) **does not** have its **existence** managed by the object (class)
- The associated object (member) **may or may not know about the existence** of the object (class)

Implementing Association

- Associations are a **broad type** of relationship
- They can be implemented in many different ways
  - Associations are implemented **using pointers**

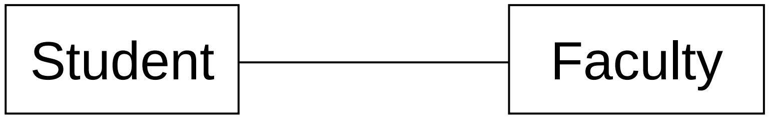


```
class A{//associated object
    private:
        //private members
    public:
        A(){
        }
};

class B{
    private:
        A* a;
    public:
        //constructors and member functions
};
```



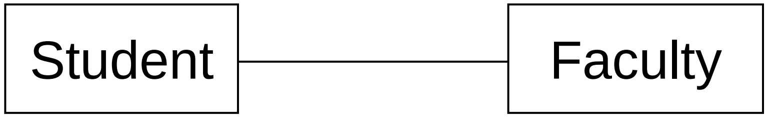
```
class A{//associated object
    private:
        B* b;
        //private members
    public:
        A(){
        }
};

class B{
    private:
        A* a;
    public:
        //constructors and member functions
};
```

Student-Faculty association - example



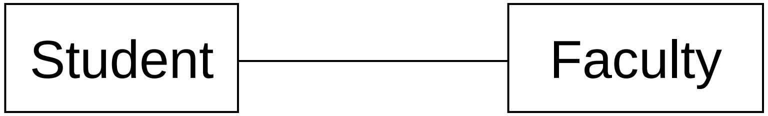
```
class Student
{
    private:
        string s_name;
        int n_faculty;
        Faculty* faculty[5]; //student can register with five faculty members only

        //this is kept private so student cannot add faculty instead faculty add student
        //addStudent function in Faculty class is public
        void addFaculty( Faculty& faculty);
    public:
        Student(const std::string& name): n_faculty{0}, s_name{ name }
        {}

        int getFacultys();

        void printFaculty();

        const std::string& getName() const { return s_name; }

        //because it need to access add faculty
        friend void Faculty::addStudent(Student& student);
};
```

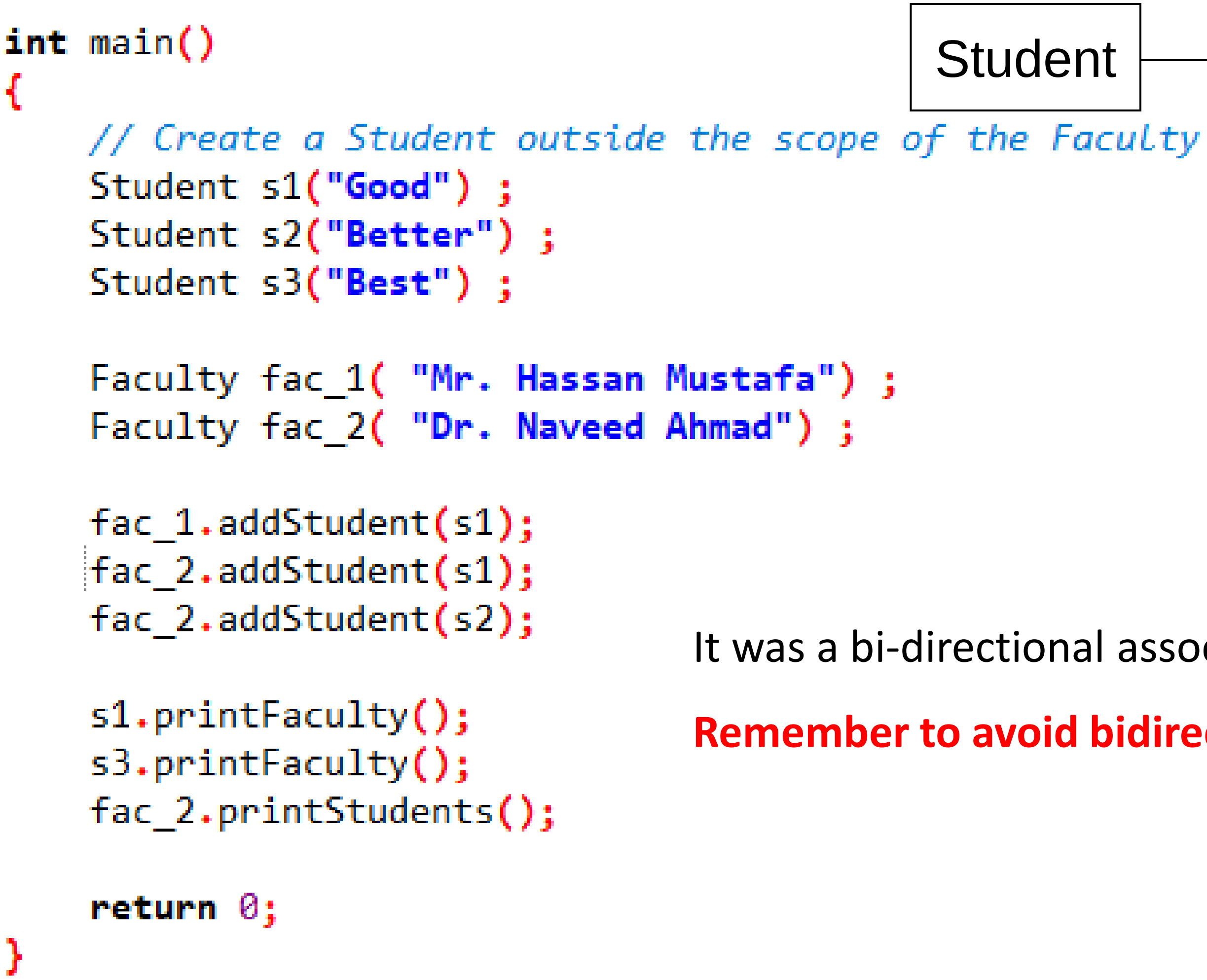
Student-Faculty association - example



```
class Faculty
{
    private:
        string f_name;
        int n_students;//total number of students a faculty can have
        Student* student[10];//faculty can have no more than 10 students

    public:
        Faculty(const std::string& name) : n_students{0}, f_name{name}
        {}

        int getStudents(){
            return n_students;
        }

        const string& getName() const {
            return f_name;
        }

        void printStudents();
        void addStudent(Student& student);

};
```

Student-Faculty association - example



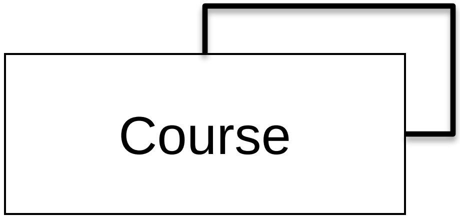

```
int main()
{
    // Create a Student outside the scope of the Faculty
    Student s1("Good") ;
    Student s2("Better") ;
    Student s3("Best") ;

    Faculty fac_1( "Mr. Hassan Mustafa") ;
    Faculty fac_2( "Dr. Naveed Ahmad") ;

    fac_1.addStudent(s1);
    fac_2.addStudent(s1);
    fac_2.addStudent(s2);

    s1.printFaculty();
    s3.printFaculty();
    fac_2.printStudents();

    return 0;
}
```

It was a bi-directional association

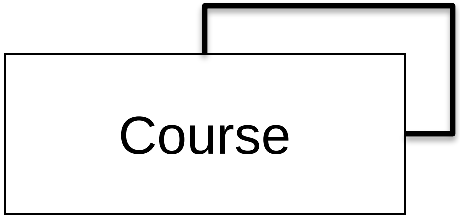
Remember to avoid bidirectional associations!!!

Reflexive association

- When **objects** have a **relationship** with **other objects** of the **same type**
  - Consider a course class

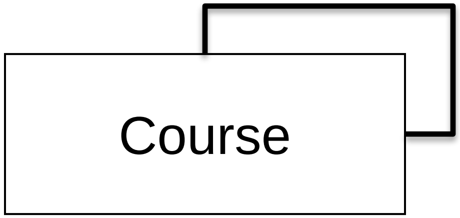


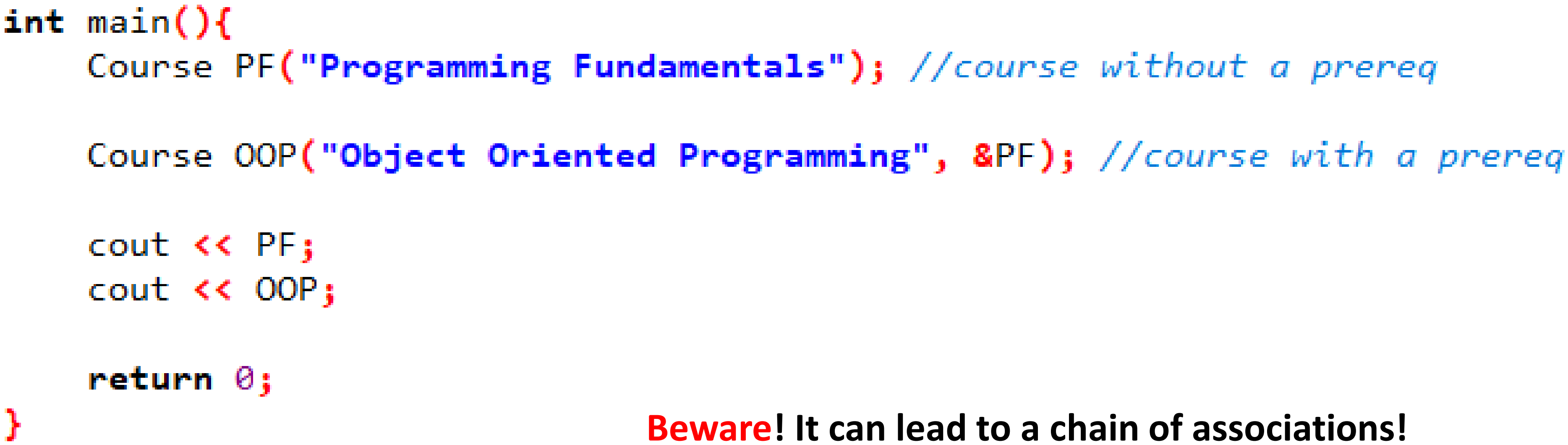
Course - Example



```
class Course
{
    public:
        string name;
        const Course *prereq;//reflexive association

    public:
        //constructor to initialize a course
        Course(const string &name, const Course *prereq = NULL)
        {
            this->name = name;
            this->prereq = prereq;
        }

        friend ostream& operator<< (ostream&, Course&);
};
```

Course - Example



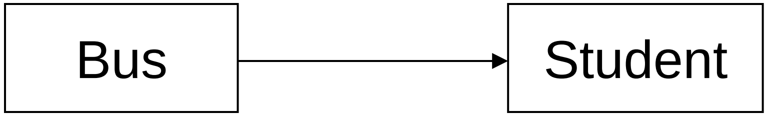


```
int main(){
    Course PF("Programming Fundamentals"); //course without a prereq

    Course OOP("Object Oriented Programming", &PF); //course with a prereq

    cout << PF;
    cout << OOP;

    return 0;
}
```

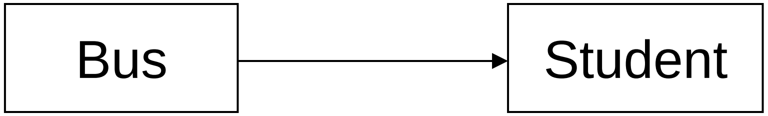
Beware! It can lead to a chain of associations!

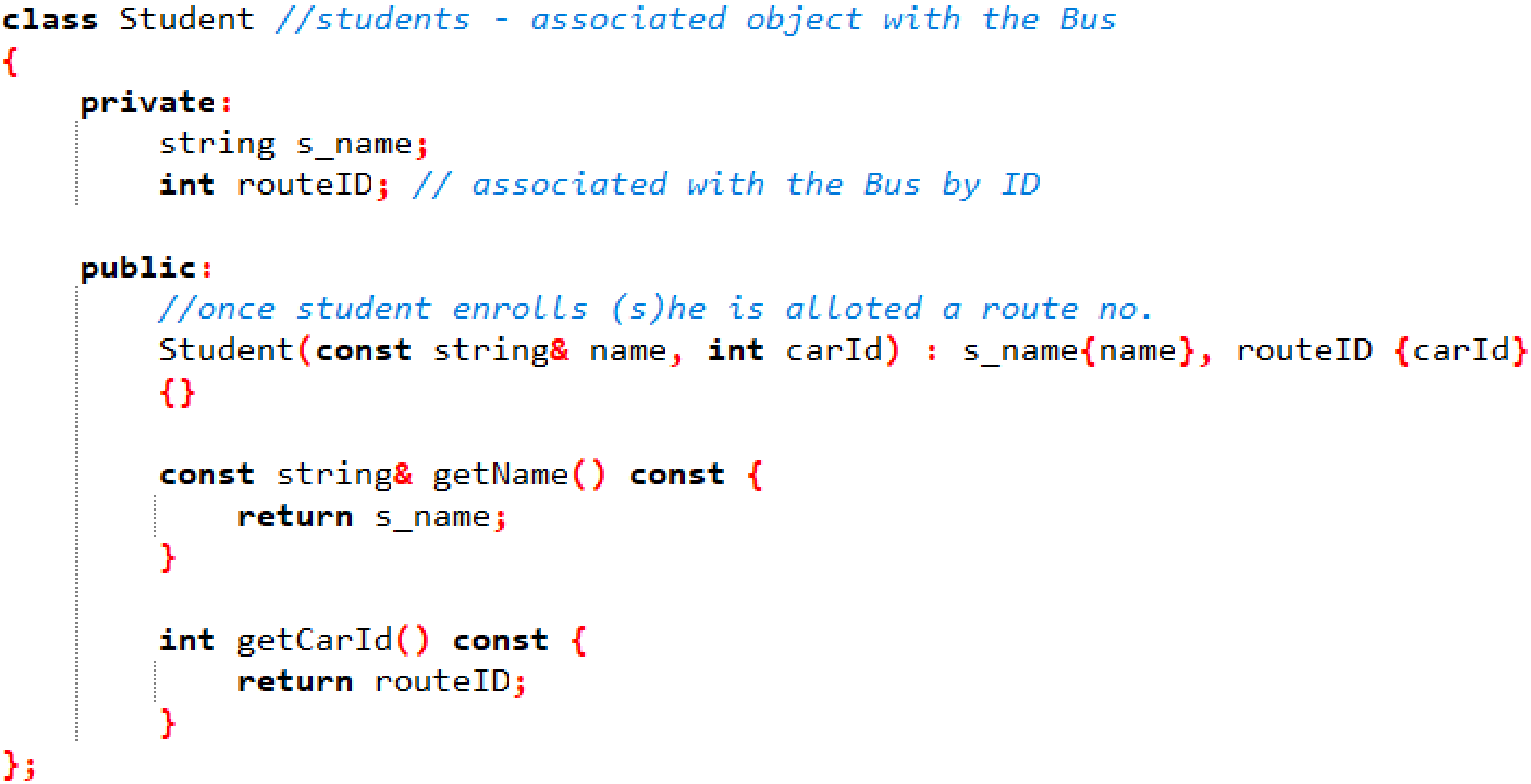
Indirect association

- In an **association** using **pointers/reference** is **not strictly required**.
  - **Any kind of data** that allows you to **link two objects** together suffices.



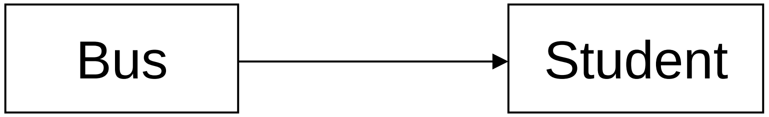
Student-Bus - Example





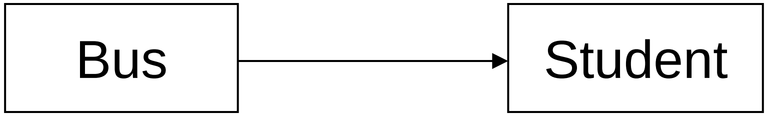
```
class Student //students - associated object with the Bus
{
    private:
        string s_name;
        int routeID; // associated with the Bus by ID

    public:
        //once student enrolls (s)he is alloted a route no.
        Student(const string& name, int carId) : s_name{name}, routeID {carId}
        {}

        const string& getName() const {
            return s_name;
        }

        int getCarId() const {
            return routeID;
        }
};
```

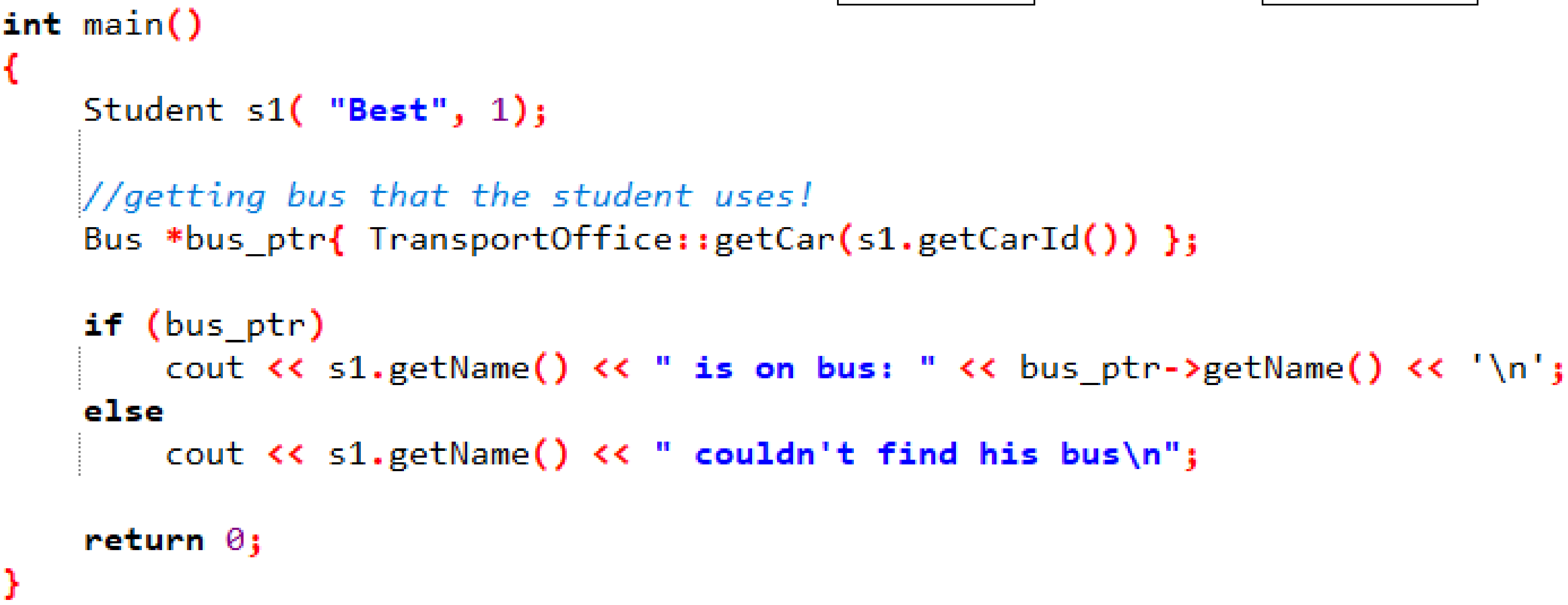
Student-Bus - Example



```
class Bus
{
    private:
        string m_name;
        int n_route;

    public:
        Bus(const string& name, int id): m_name{name}, n_route{id}
        {}

        const string& getName() const {
            return m_name;
        }

        int getId() const {
            return n_route;
        }
};
```

Student-Bus - Example

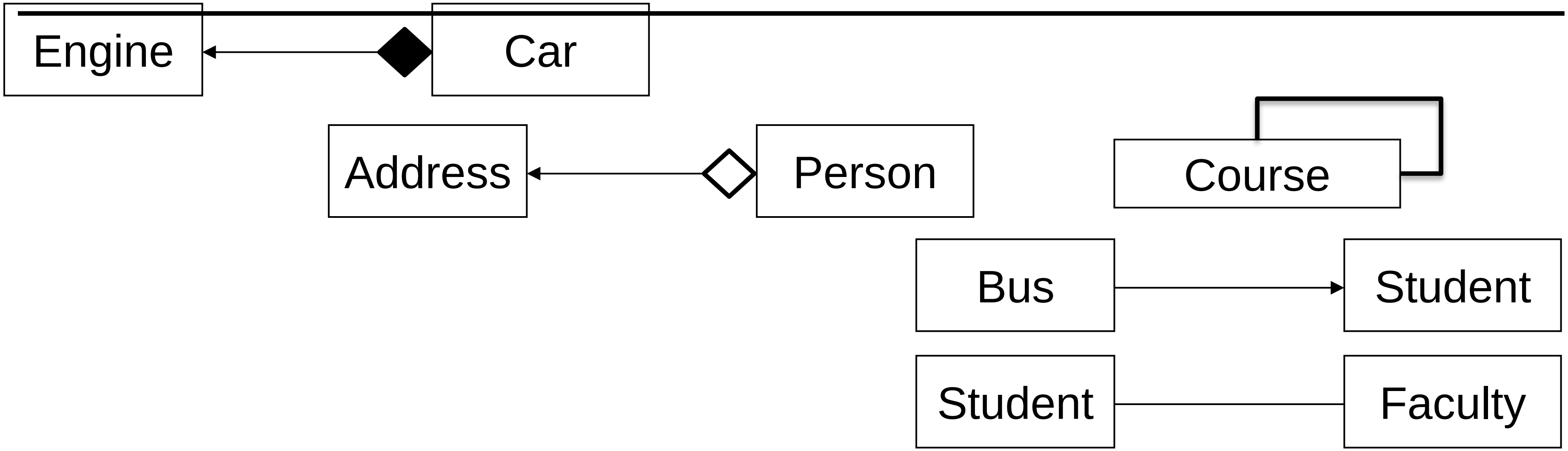




```
int main()
{
    Student s1( "Best", 1);

    //getting bus that the student uses!
    Bus *bus_ptr{ TransportOffice::getCar(s1.getCarId()) };

    if (bus_ptr)
        cout << s1.getName() << " is on bus: " << bus_ptr->getName() << '\n';
    else
        cout << s1.getName() << " couldn't find his bus\n";

    return 0;
}
```

Composition vs Aggregation vs Association

Property	Composition	Aggregation	Association
Relationship type	Whole/part	Whole/part	Otherwise unrelated
Members can belong to multiple classes	No	Yes	Yes
Members existence managed by class	Yes	No	No
Directionality	Unidirectional	Unidirectional	Unidirectional or bidirectional
Relationship verb	Part-of	Has-a	Uses-a


Engine
Car
Address
Person
Course
Bus
Student
Student
Faculty