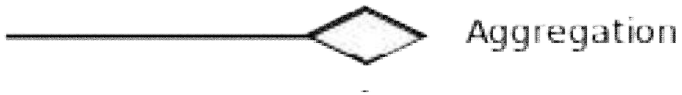
Class/Object Relationships

CS(217) Object Oriented Programming

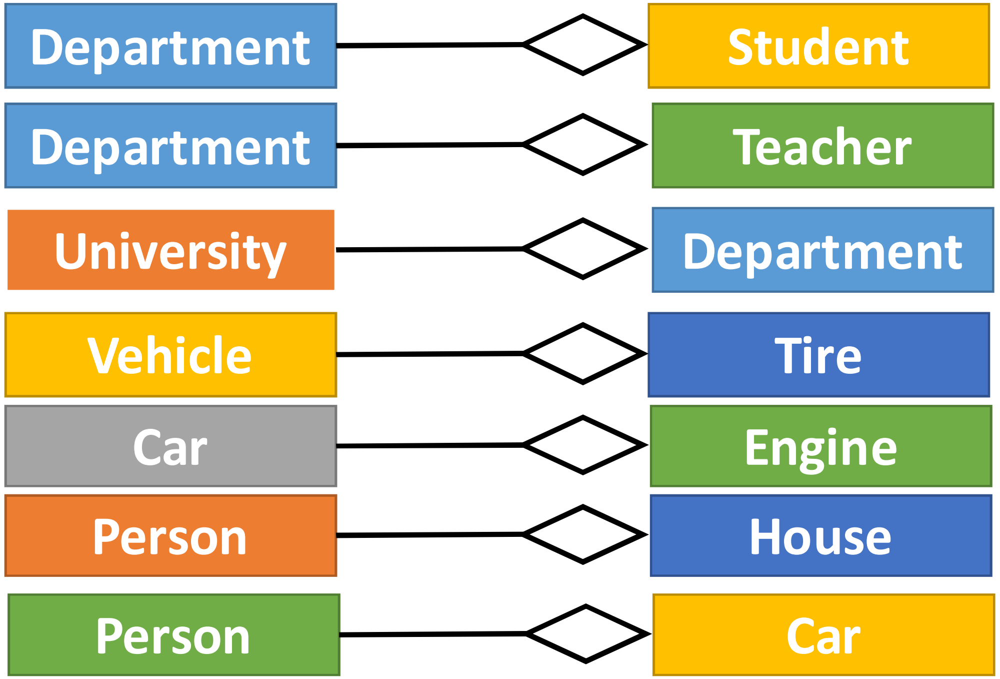
Aggregation (has-a)



- Subset of association relation where ownership is involved
- Weak relation
- Object of one class can contain object(s) of other class(s) for specific amount of time
  1. one-to-one,
  2. one-to-many
- Unidirectional object of container class knows about its parts
- Objects have independent life time (creation and destruction)

Aggregation (has-a)Examples

- One department has many students.
- A department has many teachers.
- A University has many departments.
- A vehicle has many tires.
- A car has an engine.
- A person owns a house.
- A person owns many cars.



Department ◇ Student

Department ◇ Teacher

University ◇ Department

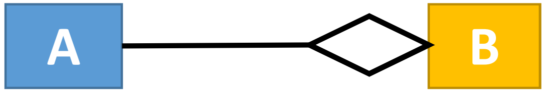
Vehicle ◇ Tire

Car ◇ Engine

Person ◇ House

Person ◇ Car

Aggregation (has-a) Implementation

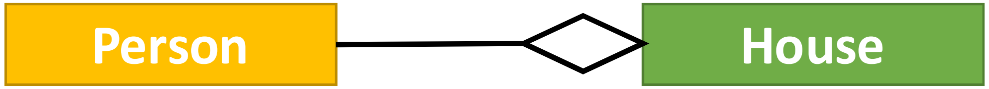


- Use a pointer to aggregate class object(s).

```
void main(){
    A a(1), a2;
    B b, b2(3);
    a.addB(&b);
    a.changeB(&b2);
    a2.add(&b);
}
```

```
class B{
    int b;
public:
    B(int b=0){ this->b=b;}
};
class A{
    int  a;
    B * objB; //pointer
public:
    A(int a=0){ this->a=a;}
    void addB(B*b){ this->objB = b;}
    void removeB(){ objB = nullptr;}
    Void changeB(B*b){ objB= b;}
    ~A(){ objB=nullptr;}
    //nothing to do with objB
};
```

Aggregation (has-a) Implementation



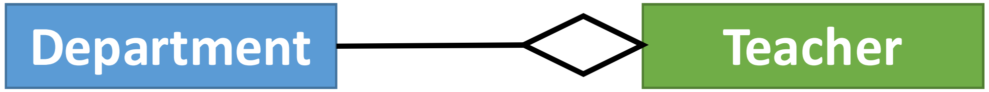
- One to one
- House pointer in person class points to aggregate class object.

```
class House{
    int hid;
public:
    House(int h=0){ this->hid=h;}
};

void main(){
    House * h  = new House(1);
    Person p(1, h);
    p.removeHouse();
    delete h;
}
```

```
class Person{
    int  pid;
    House * hptr; //pointer for house
public:
    Person(int pid, House * hptr){
         this->hptr =hptr;
         this->pid = pid;
    }
    void changeHouse(House * h){
         hptr = h;
    }
    void removeHouse(){ hptr = nullptr;}
    ~Person(){
         hptr = nullptr;
    }
};
```

Aggregation (has-a) Implementation

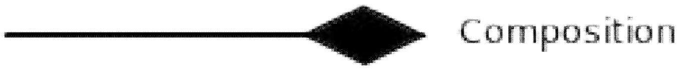


- **One to many**

```
class Teacher{
    int tid;
    char * name;
public:
    Teacher(int t=0, char*n=nullptr){
        tid=t;
        name = nullptr;
        if(n!=nullptr){
          name = new char[strlen(n)+1];
          strcpy(name, n);
        }
    }
    ~Teacher(){
        if(name != nullptr)
        delete [] name;
    }
};
```

```
class Department{
    int  did, noofteachers, current;
    Teacher ** tList; //pointers list
public:
    Department(int id = 0, int noofteachers = 10){
        this->noofteachers= noofteachers;
        tList = new Teacher * [noofteachers];
        current = 0;
    }
    void AddTeacher(Teacher * t){ tList[current++] = t; }
    void RemoveTeacher(int tid);
    ~Department(){
        for(int i=0; i< noofteachers;i++}
            tList[i] = nullptr;
        delete[] tlist;
    }
};
```

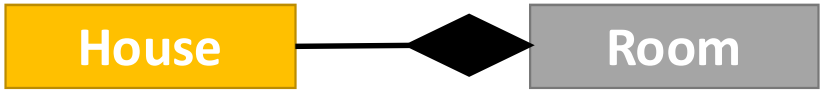
Composition (whole-part)



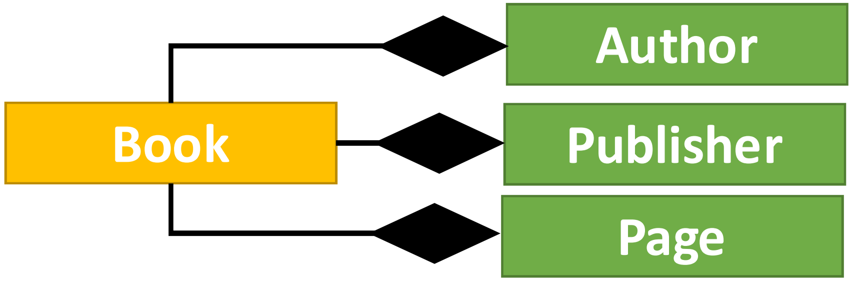
- Subset of aggregation relation where ownership is involved
- Strong relation
- Object of one class can contain object(s) of other class(s) for lifetime
  1. one-to-one,
  2. one-to-many
- Unidirectional object of container class knows about its parts
- Objects have dependent life time (creation and destruction)
  - When whole destroy part is also destroyed
  - Creation and destruction of part is controlled by whole
  - Part object can belong only to one whole class

Composition (whole-part) Examples

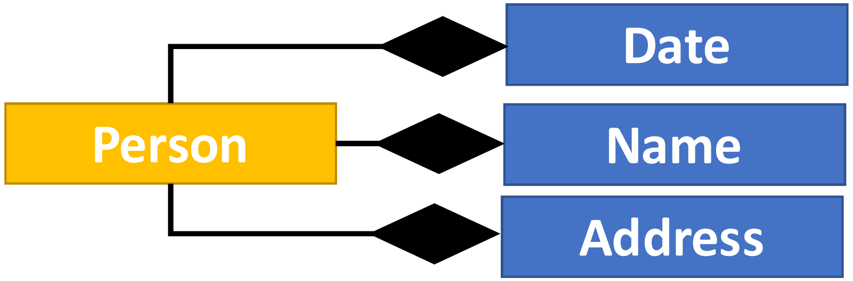
House cannot exist without rooms.



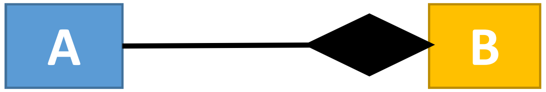
- Book cannot exist without author(s), ISBN, publisher, pages.



- Person cannot exist without name, date of birth, ID, address.



Composition (whole-part) Implementation



- Add object variable as member of class.

```
void main(){
    A a, a2(3, 4);
}
```

- When object of A is created object of B is created inside A too.
- When object of A is destroyed part object B is also destroyed.

```
class B{
    int b;
public:
    B(int b=0){ this->b=b;}
};
class A{
    int  a;
    B  objB; //variable
public:
    A(int a=0, int b=0):objB(b){
        this->a=a;
    }
    //call parametrized constructor of part
    ~A(){}
    //nothing to do with part destroyed
    automatically
};
```

This page is intentionally left blank

Composition (whole-part) Example

Person

- Single class person controls every thing

```
class Person{
    int  pid;
// Name
    char * fname;
    char * lname;
//Date of Birth
    int day;
    int mon;
    int year;
//Address
    char * city;
    char *country;
    int streetNo;
    int houseNo;
};
```

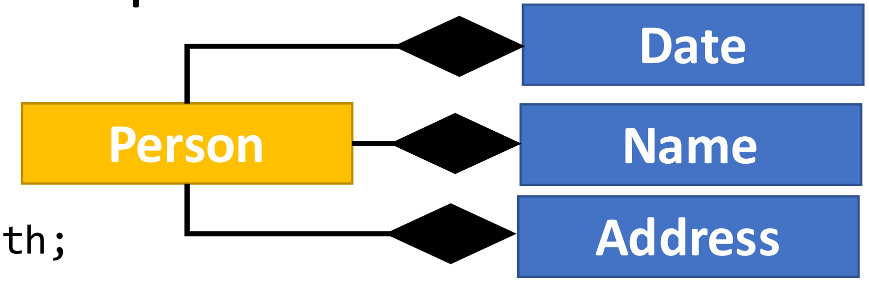
- Not scalable
- Error prone
- Not reusable in other class
- Redefine all attributes and functions separately for other classes
- For example student, doctor and teacher, patient

Composition (whole-part) Example

- Design separate classes

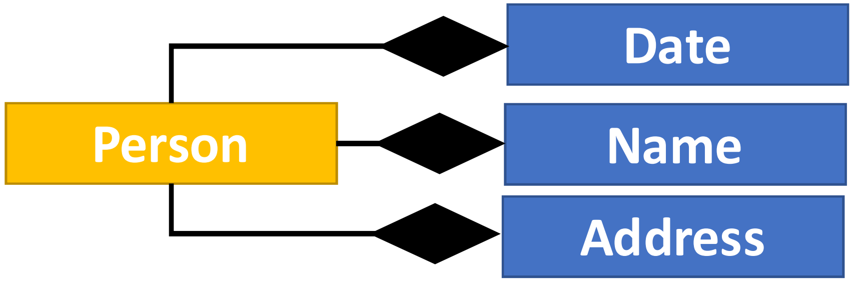
```
class name{
    char * fname;
    char * lname;
};
class date{
    int day;
    int mon;
    int year;
};
class address
    char * city;
    char *country;
    int streetNo;
    int houseNo;
};
```

```
class person{
    int  pid;
    name pname;
    date dateofBirth;
    address paddress;
};
```



- Add objects as variables in class
- Scalable
- Less Error prone
- Reusable in other classes such as student, doctor and teacher, patient
- No need to redefine all attributes and functions separately for other classes

Composition (whole-part) Example

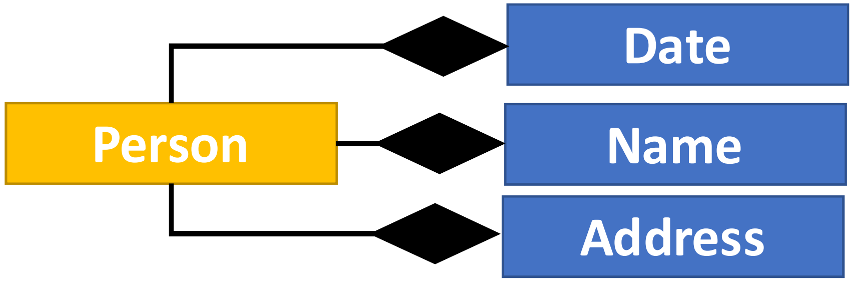


- Call functions of composed classes

```
class person{
    int  pid;
    name pname;
    date dateofBirth;
    address paddress;
public:
    person(int pid, char*fn, char*ln, int d, int m, int y, char*city, char*country,
    int street, int house)
    :pname(fn,ln), dateofBirth(d,m,y), paddress(city, country, street, house)
    {
        this->pid=pid;
    }
//call parameterized constructors for object separately
};
```

Composition (whole-part) Example

- Call functions of composed classes

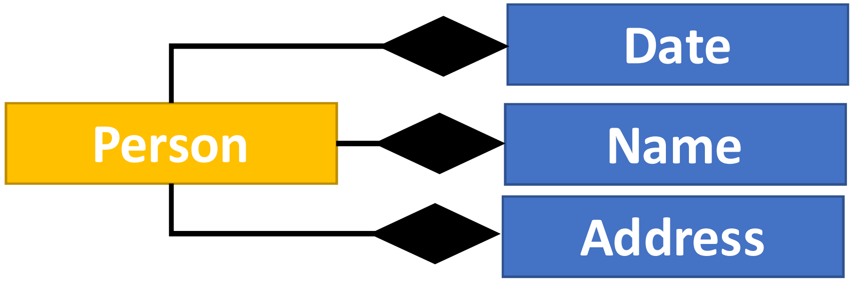


```
class person{
    int  pid;
    name pname;
    date dateofBirth;
    address paddress;
public:
    void setName(char*fn, char*ln){
            pname.setname(fn, ln);
    }
    void setDateofBirth(int d, int m, int y){
            dateofBirth.setDate(d,m,y);
    }
    void setAddress(char*city, char*country, int street, int house){
            paddress.setaddress(city, country, street, house);
    }
//Reuse functions of defined objects in person class
};
```

Composition (whole-part) Example

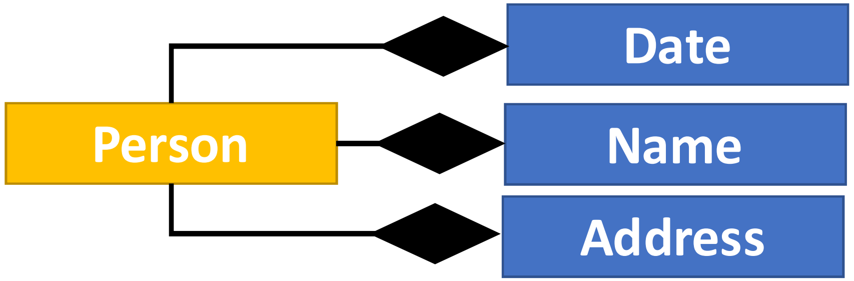
- Call functions of composed classes

```
class person{
    int  pid;
    name pname;
    date dateofBirth;
    address paddress;
public:
    char * getfirstName(){
            return pname.getfirstname();
    }
    char * getlastName(){
            return pname.getlastname();
    }
    Date getDateofBirth(int d, int m, int y){
            return dateofBirth.getDate();
    }
//Reuse functions of defined objects in person class
};
```



Composition (whole-part) Example

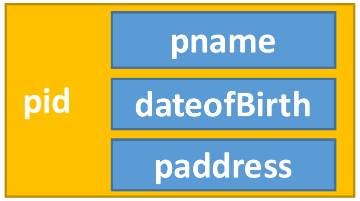
- Call functions of composed classes

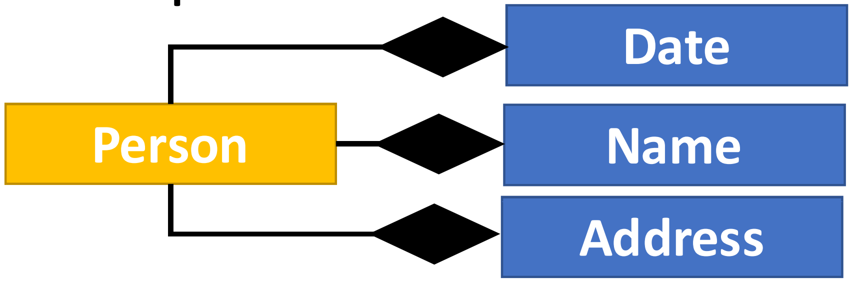


```
class person{
    int  pid;
    name pname;
    date dateofBirth;
    address paddress;
public:
    friend ostream& operator<< (ostream& , const Person&);
  //Reuse functions of defined objects in person class
};
friend ostream& operator<< (ostream& out , const Person& p){
    out<< “Person id:” << pid;
    out<< “Name:”    << pname;
    out<< “Date of Birth:”   << dateofBirth;
    out<< “Address:” << paddress;
} //Call ostream operator functions of name, date and address class
```

Composition (whole-part) Example

```
class person{
    int  pid;
    name pname;
    date dateofBirth;
    address paddress;
};
void main(){ Person p; }
```





- Calling sequence
  - **Default constructor:** in same order as defined objects in class
    1)name 2)date 3)address 4)person
  - **Destructor:** in reverse order as defined objects in class
    1)person 2)address 3)date 4)name
  - **Parametrized constructor:** called in order of member initializer syntax
    : dateofBirth(d,m,y), pname(fn,ln), paddress(city, country, street, house)
    1)date 2)name 3)address 4)person