Week 12

Class Relationships

- Association
- Aggregation
- Composition

Association

- Association is a simple structural connection or channel between classes and has a relationship where all objects have their own lifecycle and there is no owner.
- Associations are connections between peer objects, which allows objects to call each others functions Implemented as object references (i.e., class-scope variables that reference to other objects)

Association

- Associated objects may not be permanent nor must they be created at the same time (i.e., they have separate existences)
- If a Class A is associated with Class B, then A has a pointer to B object as a data member.
- Class A only has shallow version of Class B object.

Implementation of Association

- Person knows his address
- Address does not know anything about who is living there.
- From implementation perspective,
  - Person has a pointer to an Address object as data member.

One-Way Association (Person-Address)

```
class Person {
string Name;
Address *addr;
int Age;

public:
Person(){..}
~Person{..}
void setAddress(Address* a)
{ //Shallow Copy
addr = a;
}
};
```

```
class Address {
string Street;
long postalCode;
string Area;
.....

}
```

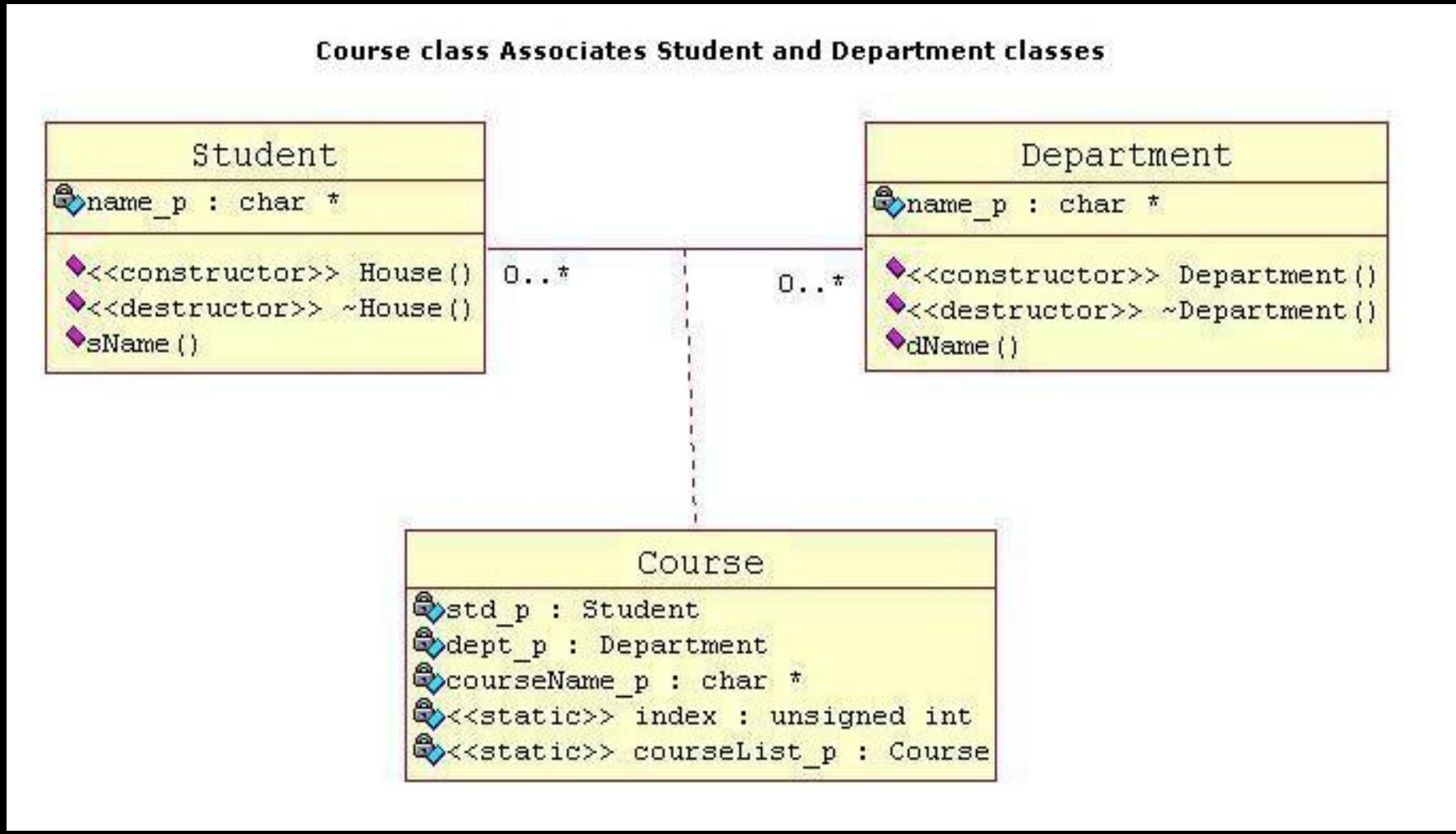
Implementation of Association

Two-Way Association

```
class Contractor
{
private:
string Name;
Project *MyProject;
…


};
```

```
class Project
{
string Name;
Contractor *person;


….


};
```

Ternary Association



Ternary Association

For complete Example:

https://www.go4expert.com/articles/association-aggregation-composition-t17264/

Composition

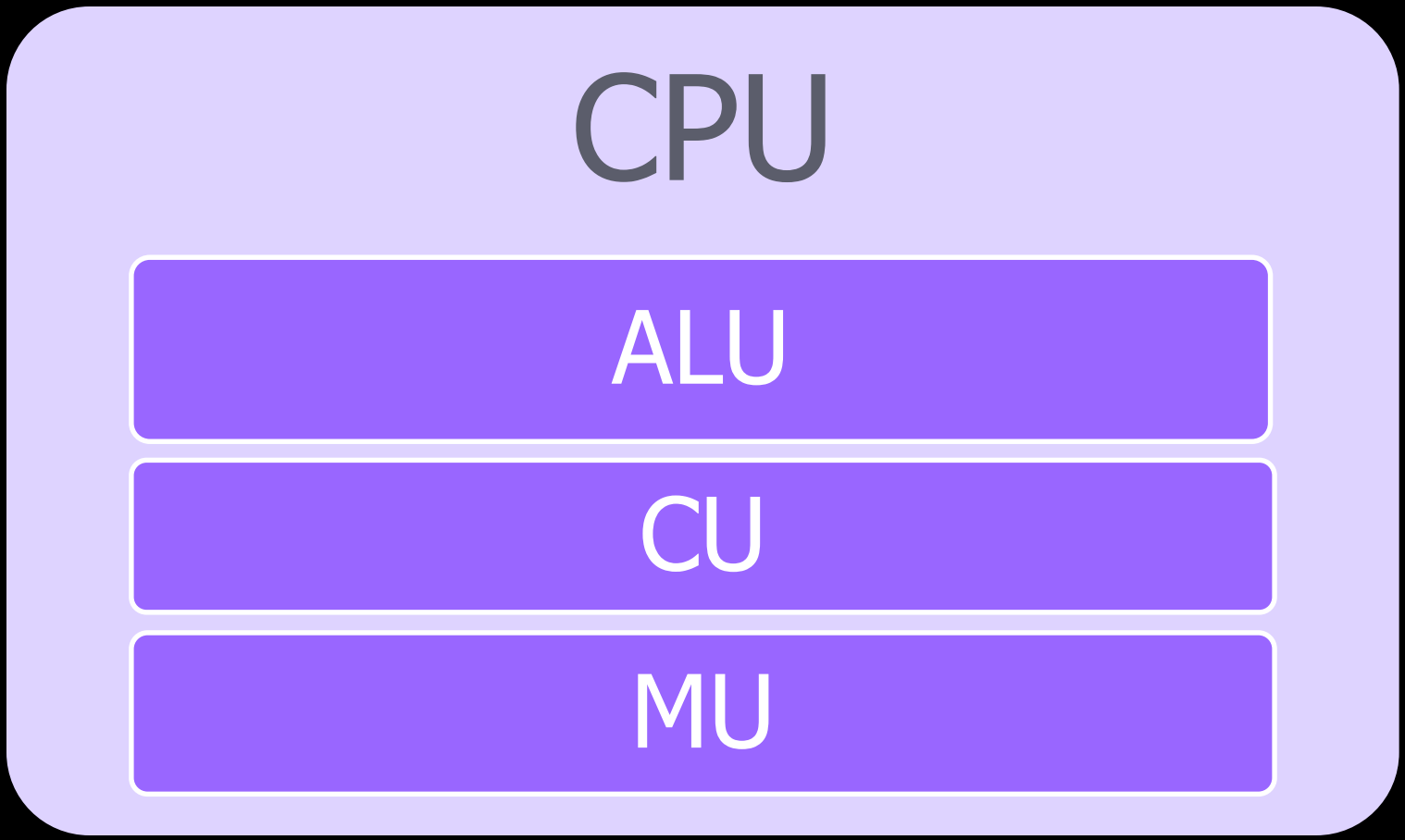
- Creating objects of one class inside another class
- When an object contains other object, if the contained object cannot exist without the existence of container object, then it is called composition
- *"Has a"* relationship:
  - Bird has a beak
  - Student has a name
  - House has rooms

Composition

Implementation details:

1. Typically use objects of other class as normal member variables.
2. Can use pointer values if the composition class automatically handles allocation/deallocation
3. Responsible for creation/destruction of subclasses
4. A has a B object as data Member.

Composition-Example 1



Implementation of Composition.Cpp

Composition-Example 2

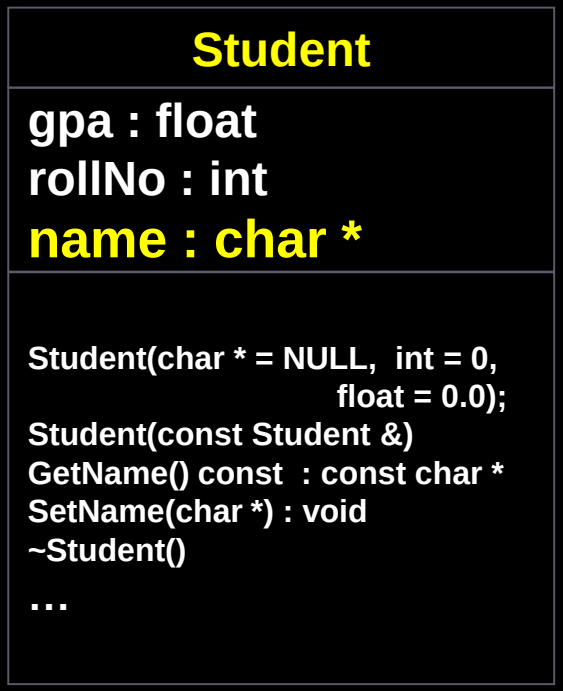
Student has a Name.

class Student {	class String {
String Name;	…
….	
};	};

Read the complete Example Yourself.
Slides 31 - 55

Composition-Example 2

Consider the following implementation of the student class:



Composition

```
class Student{
private:
   float gpa;
   char * name;
   int rollNumber;
public:
   Student(char * = NULL, int = 0,
                  float = 0.0);
   Student(const Student & st);
   const char * GetName() const;
   ~Student();
   …
};
```

Composition

```
Student::Student(char * _name, int roll,
                                 float g){
    cout << "Constructor::Student..\n";
    if (!_name){
          name = new char[strlen(_name)+1];
          strcpy(name,_name);
    }
    else name = NULL;
    rollNumber = roll;
    gpa = g;
}
```

Composition

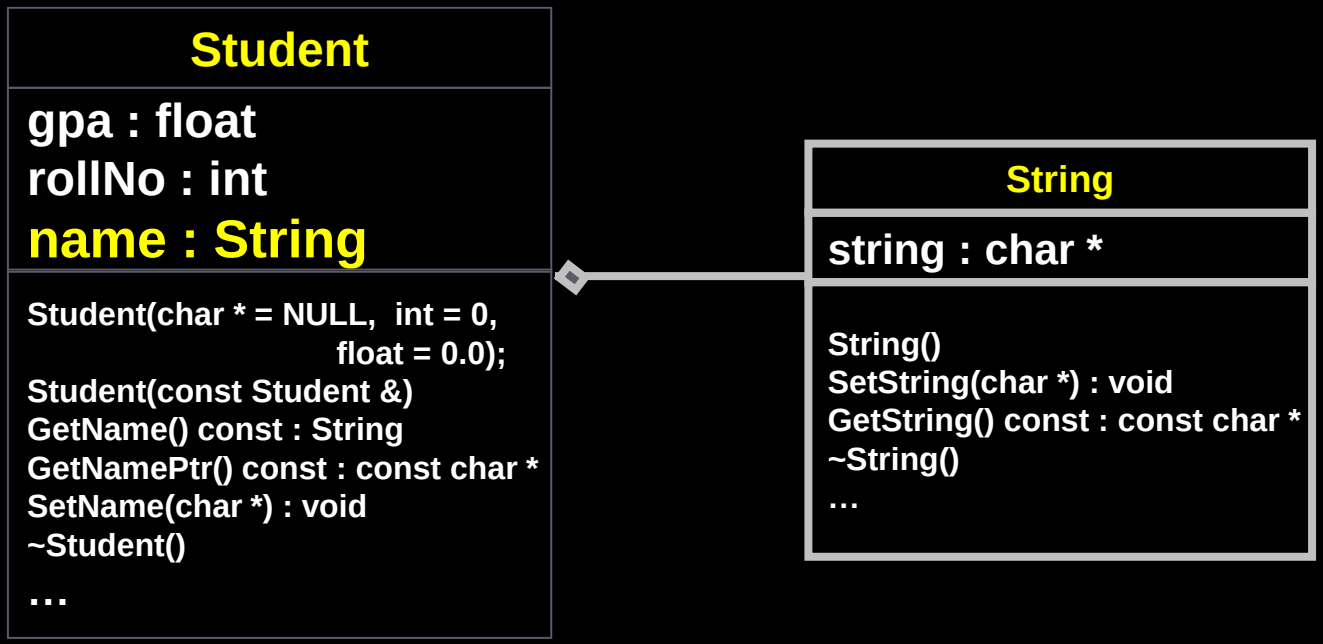
```
Student::Student(const Student & st){
    if(str.name != NULL){
        name = new char[strlen(st.name) + 1];
        strcpy(name, st.name);
    }
    else name = NULL;
    rollNumber = st.roll;
    gpa = st.g;
}
```

Composition

```
const char * Student::GetName(){
    return name;
}

Student::~Student(){
    delete [] name;
}
```

Composition

Conceptual notation:



Composition

```
class String{
  private:
     char * ptr;
  public:
     String();
     String(const String &);
     void SetString(char *);
     const char * GetString() const;
     ~String()
     …
};
```

Composition

```
String::String(){
    cout << "Constructor::String..\n";
    ptr = NULL;
}

String::String(const String & str){
    if(str.ptr != NULL){
     string = new
char[strlen(str.ptr)+1];
     strcpy(ptr, str.ptr);
    }
    else ptr = NULL;
}
```

Composition

```
void String::SetString(char * str){
    if(ptr != NULL){
        delete [] ptr;
        ptr = NULL;
    }
    if(str != NULL){
        ptr = new
char[strlen(str)+1];
        strcpy(ptr, str);
    }
}
```

Composition

```
const char * String::GetString()const{
    return ptr;
}
String::~String(){
    delete [] ptr;
    cout <<"Destructor::String..\n";
}
```

Composition

```
class Student{
private:
  float gpa;
  int rollNumber;
  String name;
public:
  Student(char* =NULL, int=0,float=0.0);
  Student(const Student &);
  void SetName(const char *);
  String GetName() const;
  const char * GetNamePtr const();
  ~Student();
  …
};
```

Composition

```
Student Student(char * _name,
                int roll,  float g){
    cout <<"Constructor::Student..\n";
    name.SetString(_name);
    rollNumber = roll;
    gpa = g;
}
```

Composition

```
Student::Student(const Student & s){
    name.Setname(s.name.GetString());
    gpa = s.gpa;
    rollNo = s.rollNo;
}

const char * Student::GetNamePtr() const{
     return name.GetString();
}
```

Composition

```
void Student::SetName(const char * n){
    name.SetString(n);
}

Student::~Student(){
    cout <<"Destructor::Student..\n";
}
```

Composition

Main Function:

```
void main(){
      Student aStudent("Fakhir", 899,
                                    3.1);
      cout << endl;
      cout << "Name:"
           << aStudent.GetNamePtr()
           << "\n";
}
```

Composition

- Output:

```
Constructor::String..
Constructor::Student..

Name:  Fakhir
Destructor::Student..
Destructor::String..
```

Composition

- Constructors of the sub-objects are always executed before the constructors of the master class

- Example:
  - Constructor for the sub-object **`name`** is executed before the constructor of **`Student`**

Composition

```
Student::Student(char * n,
                 int roll, float g){
   cout <<"Constructor::
                 Student..\n";
   name.SetString(n);
   rollNumber = roll;
   gpa = g;
}
```

Composition

- To assign meaningful values to the object, the function **`SetString`** is called explicitly in the constructor
- This is an overhead
- Sub-object **`name`** in the **`student`** class can be initialized using the constructor
- "*Member initialization list*" syntax is used

Composition

- Add an overloaded constructor to the **`String`** class defined above:

```
class String{
   char *ptr;
public:
   String();                    } //String(char * = NULL);
   String(char *);              }
   String(const String &);
   void SetName(char *);
   ~String();
   …
};
```

Composition

```
String::String(char * str){
    if(str != NULL){
          ptr = new char[strlen(str)+1];
          strcpy(ptr, str);
   }
   else ptr = NULL;
   cout << "Overloaded
          Constructor::String..\n";
  }
```

Composition

- Student class is modified as follows:

```
class Student{
private:
    float gpa;
    int rollNumber;
    String name;
public:
    …
    Student(char *=NULL, int=0, float=0.0);
};
```

Composition

- Student class continued:

```
Student::Student(char * n,int roll,
                 float g): name(n){
    cout << "Constructor::Student..\n";
    rollNumber = roll;
    gpa = g;
}
```

Composition

Main Function:

```
int main(){
      Student aStudent("Fakhir", 899,
                                    3.1);
      cout << endl;
      cout << "Name:"
           << aStudent.GetNamePtr()
           << endl;
      return 0;
}
```

Composition

- Output:

```
Overloaded Constructor::String..
Constructor::Student..

Name:  Fakhir
Destructor::Student..
Destructor::String..
```

Aggregation

Composition vs. Aggregation

- Aggregation is a *weak relationship*
- **Aggregation** is a specialize form of Association where all object have their own lifecycle but there is a ownership like parent and child.
- We can think of it as "has-a" relationship.

Aggregation

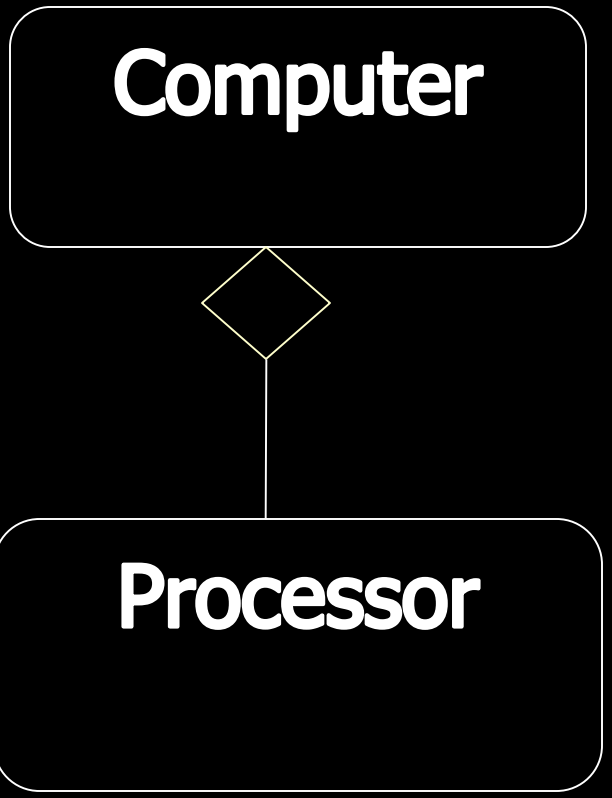
Implementation details:

- Typically use pointer variables that points to an object that lives outside the scope of the aggregate class
- Can use reference values that point to an object that lives outside the scope of the aggregate class
- **Not** responsible for creating/destroying subclasses
- Has a pointer to Object and data of pointing object is **deep copied** in that pointer.

Aggregation

- In aggregation, a pointer or reference to an object is created inside a class
- The sub-object has a life that is **NOT** dependant on the life of its master class
- e.g:
  - Chairs can be moved inside or outside at anytime
  - When Room is destroyed, the chairs may or **may not** be destroyed

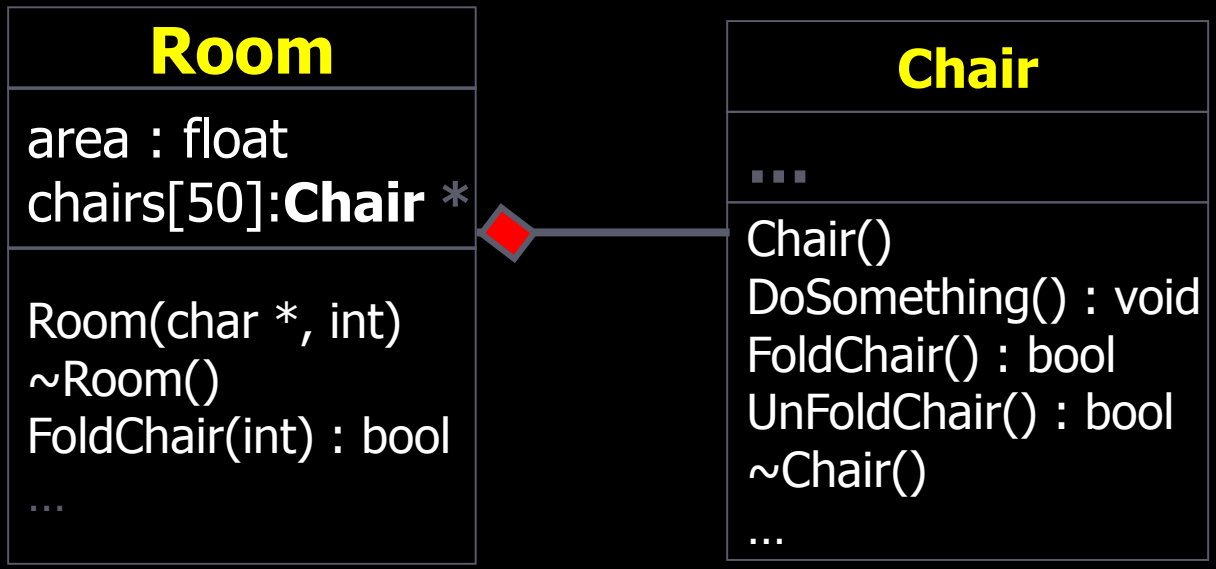
Aggregation-Example 1



Implementation of Aggrgation.Cpp

Aggregation-Example 2

- Room has Chairs
- *Read Complete Example Yourself.*
  - *Slide 60 - 64*



Aggregation

```
class Room{
private:
   float area;
   Chair * chairs[50];
Public:
   Room();
   void AddChair(Chair *, int chairNo);
   Chair * GetChair(int chairNo);
   bool FoldChair(int chairNo);
   …
};
```

Aggregation

```
Room::Room(){
   for(int i = 0; i < 50; i++)
      chairs[i] = NULL;
}
void Room::AddChair(Chair *
         chair1, int chairNo){
   if(chairNo >= 0 && chairNo < 50)
         chairs[chairNo] = chair1;
}
```

Aggregation

```
Chair * Room::GetChair(int chairNo){
   if(chairNo >= 0 && chairNo < 50)
         return chairs[chairNo];
   else
         return NULL;
}

bool Room::FoldChair(int chairNo){
 if(chairNo >= 0 && chairNo < 50)
   return chairs[chairNo]->FoldChair();
 else
   return false;
}
```

Aggregation

```
int main(){
    Chair ch1;
    {
        Room r1;
        r1.AddChair(&ch1, 1);
        r1.FoldChair(1);
    }
    ch1.UnFoldChair(1);
    return 0;
}
```

Aggregation or Composition? Lake has slide..

Aggregation or Composition? Browser has tabs…

Your Turn

- Implement only one example of Association, Aggregation and Composition taken from your own project.