

OOP

Lecture 18

14/12/2021

Aggregation :

Aggregation is basically creating a new class by using an existing class by taking the objects of the existing class as a data member of the new class. We studied that, to do aggregation, there must be a **HAS-A** relationship between the classes. Or we can also say “Something is a part of something”. But, there are two types of aggregation, Strong Aggregation and Weak Aggregation.

Strong Aggregation :

Strong Aggregation between classes exists when something is a necessary part of something or we can say that there must exist an object otherwise the class is not completed. In real life, we can say that Engine is the necessary part of the Car. There must exist an Engine to complete the car. So, there is a strong aggregation between the car and the engine.

Weak Aggregation :

Weak Aggregation between classes exists when something can be a part of something, or we can say that it is not a necessary part but it exists in the class to increase functionality of the class. In real life, we can say that AC can be a part of the car, or a speaker can be a part of the car. There is always a slot given in the car for such additions but it does not affect the basic functionality of the car. So, there is a weak aggregation between the car and the AC.

Making Strong Aggregation :

Strong Aggregation between two classes can be done by simply taking the object of one class as the data member of the other class. Like,

```
class SampleA {
```

```
    _____
```

```
    _____
```

```
    public:
```

```
    _____
```

```
    _____
```

```
};
```

```
class SampleB {
```

```
    SampleA obj;    //Taking an object of SampleA class as
```

```
    datamember
```

```
    public :
```

```
    _____
```

```
};
```

Here, we created the object of the SampleA class in the SampleB class. The object of the SampleA class is dependent on the SampleB class. It will be created when we create an object of the SampleB class and also will be destroyed with it. We cannot explicitly create or delete the data members of the class. This is an example of strong Aggregation, when we simply take objects as data members. By default, it is strong, meaning the object of another class will be created and destroyed with the object of the new class.

Sample Program :

```
class Point
{
    int x, y;
public:
    Point(int x=0, int y=0)
    {
        this->x = x;
        this->y = y;
        cout << "Point Constructor\n";
    }

    ~Point(){    cout << "Point Destructor\n";    }
};

class Triangle
{
    Point p1, p2, p3; //Strong aggregation by default
public:
    Triangle()
    {
        cout << "Triangle Constructor\n";
    }
    ~Triangle(){    cout << "Triangle Destructor\n";    }
};

int main()
{
    Triangle t;
    return 0;
}
```

```
Select C:\Users\umair\Downloads\C
Point Constructor
Point Constructor
Point Constructor
Triangle Constructor
Triangle Destructor
Point Destructor
Point Destructor
Point Destructor
-----
Process exited after 10.8
Press any key to continue
```

To make Weak Aggregation, there must exist a pointer to class object instead of the static pointer. However, we can do both strong and weak aggregation with the pointer. A pointer represents an empty slot which can be filled with the pointer at run time. In short form, Strong aggregation can be done by both methods i.e with or without using pointers, however, weak aggregation can not be done without using the pointers.

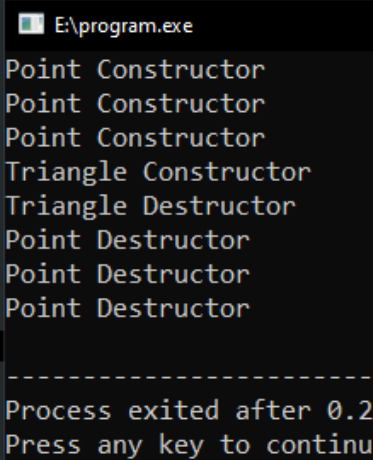
Strong Aggregation using Pointers :

Strong Aggregation can also be done using pointers, Pointers are used to make dynamic objects and dynamic arrays of objects. If we define pointer to the class as our data member and make dynamic objects only in the constructors and delete the object in the destructor, somehow, we would be implementing the same case that is the object of the existing class will be created and destroyed only when the

object of new class is created and destroyed. So, it will also be the strong aggregation.

Sample Program :

```
int x, y;
public:
Point(int x=0, int y=0){
    this->x = x;
    this->y = y;
    cout << "Point Constructor\n";
}
~Point(){    cout << "Point Destructor\n";    }
};
//Now we are implementing strong aggregation using pointers
class Triangle{
    Point *p1, *p2, *p3;
public:
    Triangle(){
        p1 = new Point;
        p2 = new Point;
        p3 = new Point;
        cout << "Triangle Constructor\n";
    }
    ~Triangle(){
        cout << "Triangle Destructor\n";
        delete p1;
        delete p2;
        delete p3;
    }
};
int main(){
    Triangle t;
    return 0;
}
```



Here, we created dynamic objects in the constructors and deleted them in the destructors, so here the objects and class makes strong aggregation.

Now we are going to use a Dynamic array for strong Aggregation, if we declare the dynamic array in the constructor and only delete the array in the destructor, it will be a strong aggregation.

Note : There must exist a non-parameterized constructor or a constructor with default values to make dynamic array of objects.

Sample Program :

```
class Point{
    int x, y;
public:
    Point(int x=0, int y=0){
        this->x = x;
        this->y = y;
        cout << "Point Constructor\n";
    }
    ~Point(){    cout << "Point Destructor\n";    }
};

class Triangle{
    Point *p; //Dynamic array
    int size;
public:
    Triangle(int size = 10){
        this->size = size;
        p = new Point[size];
        cout << "Triangle Constructor\n";
    }
    ~Triangle(){
        delete []p;
        cout << "Triangle Destructor\n";
    }
};

int main(){
    Triangle t(3);
    return 0;
}
```

E:\program.exe

Point Constructor
Point Constructor
Point Constructor
Triangle Constructor
Point Destructor
Point Destructor
Point Destructor
Triangle Destructor

Process exited after 0.127 se
Press any key to continue . .

Weak Aggregation :

Weak Aggregation means the object can be a part of the class. It means that there must be a slot free for the object every time; however, we should decide depending on the logic of our program that when and where we require the objects and when we have to delete the objects. These objects do not depend on the object of other class, these can be created and destroyed anytime in the program. In programming the empty slot represents the pointer, we can occupy this space by taking the dynamic object anytime in the program. Hence, for weak aggregation there must exist a pointer to the class. See Sample program for further understanding.

```
int x, y;
public:
Point(int x=0, int y=0){
    this->x = x;
    this->y = y;
    cout << "Point Constructor\n";
}
~Point(){ cout << "Point Destructor\n"; }
};
class Triangle{
    Point *p1, *p2, *p3;
public:
    Triangle(){
        cout << "Triangle Constructor\n";
    }
    createObjects(){
        p1 = new Point;
        p2 = new Point;
        p3 = new Point;
    }
    destroyObjects(){
        delete p1;
        delete p2;
        delete p3;
    }
    ~Triangle(){
        cout << "Triangle Destructor\n";
    }
};
int main(){
    Triangle t;
    cout << "**** Objects of point class are not created****\n";
    cout << "**** now we are creating objects of point class****\n";
    t.createObjects();
    cout << "**** Now we are going to destroy objects of point class****\n";
    t.destroyObjects();
    cout << "**** Triangle class still exit****\n";
}
```

```
E:\program.exe
Triangle Constructor
**** Objects of point class are not created****
**** now we are creating objects of point class****
Point Constructor
Point Constructor
Point Constructor
**** Now we are going to destroy objects of point class****
Point Destructor
Point Destructor
Point Destructor
**** Triangle class still exit****
Triangle Destructor

-----
Process exited after 0.08639 seconds with return value 0
Press any key to continue . . .
```

It will be best practice to always initialize the pointers with NULL in the constructor of the class, so we can apply a check (if condition) to check if the dynamic object already exists or not.

If we tried to call any member function by using the pointer to the function without declaring a dynamic object, it would throw a very famous NULL Exception Error, and our program would crash.

Below, a sample program is given, you can rewrite this program and do certain things with the program. Like, try removing the second line of the constructor i.e, p1 = p2 = p3 = NULL. When we remove this line, and call the show function before the createPoints() function, then none of the pointers will contain NULL. All of these have some garbage value, but the dynamic objects are not created yet. So, it will try to call the p1->show() function, but as the object is not created it will throw a NULL exception error and our program will crash.

```

class Triangle{
    Point *p1, *p2, *p3;
public:
    Triangle(){
        cout << "Triangle Constructor\n";
        p1=p2=p3=NULL;
    }
    createObjects(){
        p1 = new Point;
        p2 = new Point;
        p3 = new Point;
    }
    destroyObjects(){
        delete p1;
        delete p2;
        delete p3;
    }
    void show(){
        if (p1!=NULL)    p1->show();
        if (p2!=NULL)    p2->show();
        if (p3!=NULL)    p3->show();
    }
    ~Triangle(){
        cout << "Triangle Destructor\n";
        delete p1;
        delete p2;
        delete p3;
    }
};

int main(){
    Triangle t;
    t.show();
    t.createObjects();
    t.show();
    return 0;
}

```

```

E:\program.exe

Triangle Constructor
Point Constructor
Point Constructor
Point Constructor
0,0
0,0
0,0
Triangle Destructor
Point Destructor
Point Destructor
Point Destructor
-----
Process exited after 0.13
Press any key to continue

```

Class Activity :

An AC class is given with a single parameterized constructor of size of the AC, and a show function is written in the class, for more understanding, we are writing the complete AC class here,

```

class AC{
    float size;
public:
    AC(float size){
        this->size = size;
    }
}

```

```

        void show(){
            cout << "Size: " << size << " ton\n";
        }
};

```

We have to write a Hall class which will contain 4 ACs, we have to 4 functions like, installAC1, installAC2, installAC3, installAC4, to install the ACs, and when the user calls the function, the corresponding object will be created and we also have to write a show function, in which we will write the details of the AC if the AC is install otherwise we have to write AC is not installed.

Solution :

The very first thing we need to analyze is whether it is a strong aggregation or a weak aggregation. In this case, we can clearly see that our hall is not dependent on the AC class, and ACs may or may not be part of the Hall class. So, it should be weak aggregation.

We have studied earlier, for weak aggregation there must exist a pointer, so in this case we will have 4 pointers representing 4 ACs. Here, the pointer only represents the empty slots where the ACs can be installed, maybe we install all 4 ACs, maybe we install only 2 and maybe we don't install any AC. So, the very first thing we found was that we must have 4 pointers. Next, we studied that we should initially initialize all the pointer with NULL value to indicate that no new object is created yet. When we create the object it will be overwritten over the NULL value.

The Solution Code is as follow :

```

class Hall{
    AC *a1, *a2, *a3, *a4; // Taking 4 pointers of AC class
                          // we can also use array of pointers, like
                          //    AC *a[4];
                          // The only difference between these two
//will be, it can be used in a loop in constructor and show function.

```



```

public:
    Hall()
    {
        a1 = a2 = a3 = a4 = NULL; // Initializing NULL
    }

    void installAC1( float s )
    {
        a1 = new AC(s); // Creating Dynamic Object, s will be
        passed to the parameterized constructor of the AC class for ac1 object
    }

    void installAC2( float s )
    {
        a2 = new AC(s); // Creating Dynamic Object, s will be
        passed to the parameterized constructor of the AC class for ac2 object
    }

    void installAC3( float s )
    {
        a3 = new AC(s);
    }

    void installAC4( float s )
    {
        a4 = new AC(s);
    }

    void show()
    {
        if (a1 == NULL) cout << "AC 1 is not installed\n";
        else            a1->show();

        if (a2 == NULL) cout << "AC 2 is not installed\n";
        else            a2->show();

        if (a3 == NULL) cout << "AC 3 is not installed\n";
        else            a3->show();

        if (a4 == NULL) cout << "AC 4 is not installed\n";
    }

```

```

        else                a4->show();

    }

};

```

Here, the Object will be created only when the user calls the install function of the corresponding AC. and we can also define the delete functions of the objects to delete the AC objects when there is no more need. But, don't forget to check the pointer if it is already NULL, because deleting a NULL pointer also gives an error, and also again initialize the pointer with NULL after deleting to continue best programming practice.

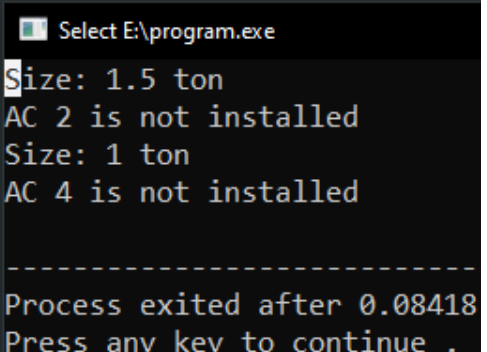
Output with main() function :

```

void installAC2(float s){
    a2 = new AC(s);
}
void installAC3(float s){
    a3 = new AC(s);
}
void installAC4(float s){
    a4 = new AC(s);
}
void show(){
    if (a1==NULL)    cout << "AC 1 is not installed\n";
    else             a1->show();
    if (a2==NULL)    cout << "AC 2 is not installed\n";
    else             a2->show();
    if (a3==NULL)    cout << "AC 3 is not installed\n";
    else             a3->show();
    if (a4==NULL)    cout << "AC 4 is not installed\n";
    else             a4->show();
}

};
int main(){
    Hall h;
    h.installAC1(1.5);
    h.installAC3(1.0);
    h.show();
    return 0;
}

```



```

Select E:\program.exe
Size: 1.5 ton
AC 2 is not installed
Size: 1 ton
AC 4 is not installed
-----
Process exited after 0.08418
Press any key to continue .

```

-THE END-

