

OOP

Lecture 17

09/12/2021

Aggregation :

Aggregation in C++ means to create a class using some existing class, i.e by taking the object(s) of the existing class as a datamember of the new class. It provides another way to reuse the class. It represents a **HAS-A** relationship association, or it has class and relationship. For Example,

- Room has Chairs
- Company has multimedia
- Student has Address

These examples contain a **HAS-A** relationship. All of these are individual classes. We take the object of the Address class as a datamember of the Person class. That's what aggregation means. We just need to ensure the **HAS-A** relationship, We can say that "Car has Engine" it is valid, but we cannot say that "Car has Road", Although car and road are related to each other but the statement "Car has Road" is not valid so aggregating these classes here is not valid.

There are 2 approaches to do aggregation :

1. Classes are already existing and we only have to reuse them.
2. Starting from scratch and building all the classes one by one and applying aggregation where required.

There are further 2 techniques while we are working from scratch.

1. **Bottom Up** : Making classes that are to be aggregated first. i.e making chair, engine, address classes etc these classes first and then making the container classes
2. **Top Down** : Classes are built only when required. It means that first we created a person class then we need the address of each person

and address also has its own attributes, so we make address class. Making the Car class first and then making the Engine class.

Sample and Example Program : Focus on the Syntax,

```
#include <iostream>

using namespace std;

class Point{
    int x, y;

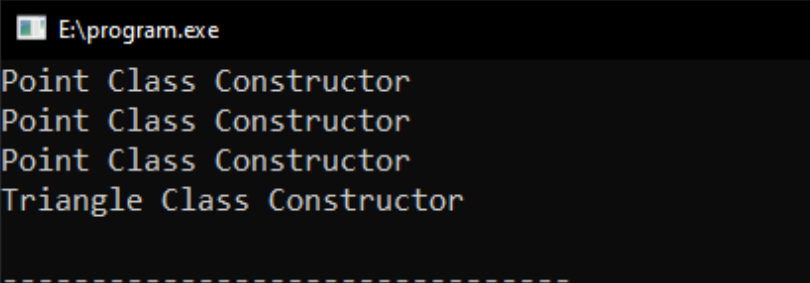
public:
    Point(){ cout << "Point Class Constructor\n"; }
};

class Triangle{
    Point p1, p2, p3;

public:
    Triangle(){ cout << "Triangle Class Constructor\n"; }
};

int main()
{
    Triangle t;

    return 0;
}
```



```
E:\program.exe
Point Class Constructor
Point Class Constructor
Point Class Constructor
Triangle Class Constructor
```

Here, we created a Point class and a Triangle class. Remember, We have to make sure we have a **HAS-A** relationship. And Triangle has Points. So, we can do aggregation here. In this program, we created three objects of the Point class in our Triangle class, as every triangle has three points, so ultimately, when we create an object of the Triangle class in main() function, it will create 4 objects, that's why four constructors are called, and a total of 6 variable, that are x and y data members of three point objects.

Now moving further, when aggregating classes, we are only interested in the public members of the class whose object is created. Here, member means both Data members (which is very rare) and Member functions. Normally, we only access the public member functions of the class. And in functions, the first function is a Constructor. If non-parameterized or default constructor is available in the class, then we don't need to do anything (Like in the above example), but if the non-parameterized constructor is not available and only parameterized constructor is available, then we have to pass some parameters to the objects of the class. And we cannot pass parameters at the time of declaration of our objects, it will give an error, like this,

```
class Triangle{
    Point p1(2, 3), p2, p3;
    private Point Triangle::p1 (2, 3)
public:
```

To pass the parameters to the Point class we need to use Initialization Lists compatibility of C++. To do so, The code is given below :

```
#include <iostream>

using namespace std;

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

class Triangle{
    Point p1, p2, p3;
public:
    Triangle() : p1(1, 1),p2(2, 2),p3(3, 3)
    {
        cout << "Triangle Class Constructor\n";
    }
};

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

```
E:\program.exe
Point Class Constructor
1,1
Point Class Constructor
2,2
Point Class Constructor
3,3
Triangle Class Constructor

-----
Process exited after 0.4564 se
Press any key to continue . .
```

We can also get these parameters from the main() function and pass them to the Point class to initialize the coordinates.

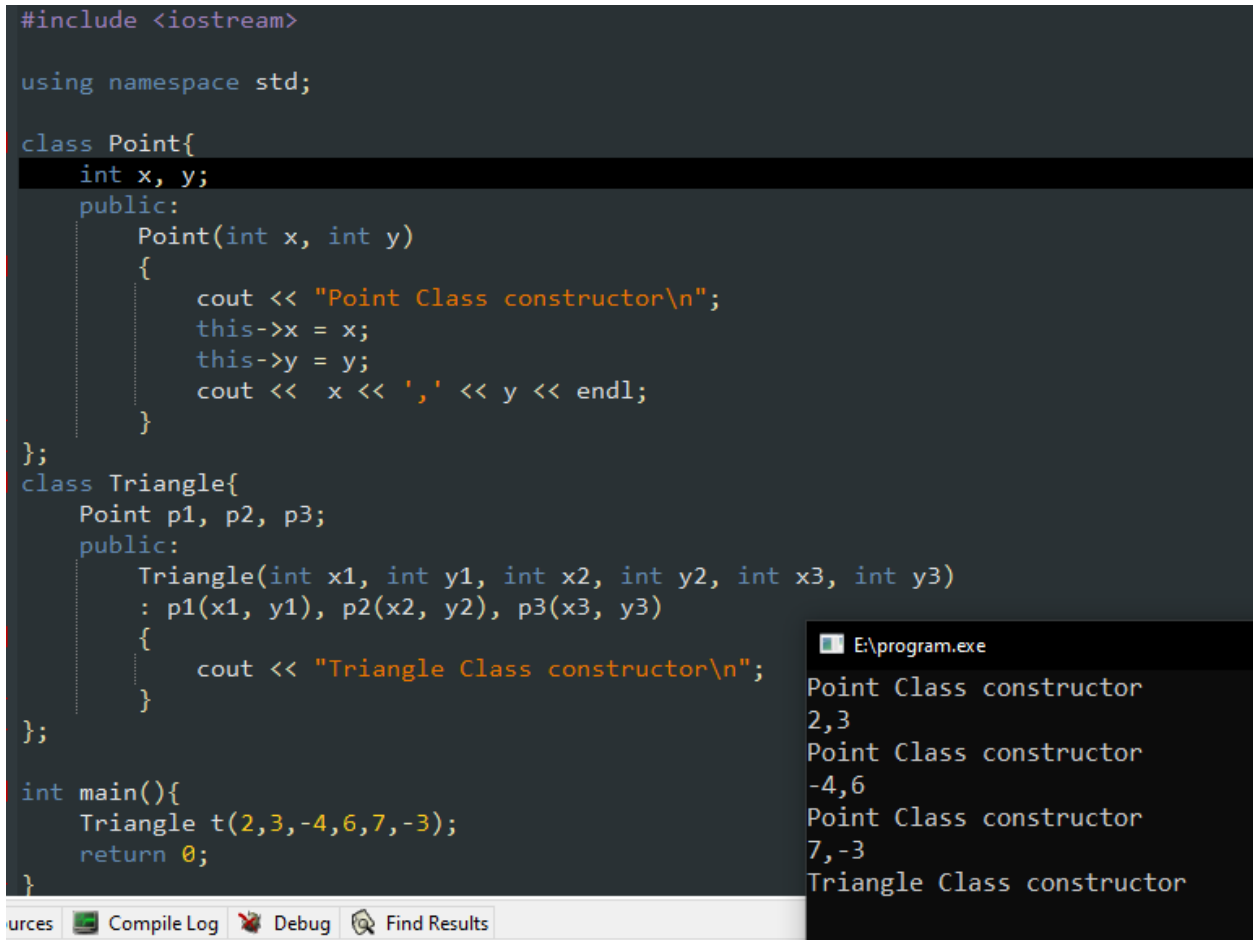
```
#include <iostream>

using namespace std;

class Point{
    int x, y;
public:
    Point(int x, int y)
    {
        cout << "Point Class constructor\n";
        this->x = x;
        this->y = y;
        cout << x << ',' << y << endl;
    }
};

class Triangle{
    Point p1, p2, p3;
public:
    Triangle(int x1, int y1, int x2, int y2, int x3, int y3)
    : p1(x1, y1), p2(x2, y2), p3(x3, y3)
    {
        cout << "Triangle Class constructor\n";
    }
};

int main(){
    Triangle t(2,3,-4,6,7,-3);
    return 0;
}
```

The image shows a C++ IDE with a code editor and a console window. The code defines a Point class with a constructor that prints the coordinates and a Triangle class that aggregates three Point objects. The main function creates a Triangle object with coordinates (2,3), (-4,6), and (7,-3). The console output shows the execution of the constructors for each Point object and the Triangle object.

```
E:\program.exe
Point Class constructor
2,3
Point Class constructor
-4,6
Point Class constructor
7,-3
Triangle Class constructor
```

Hopefully, we gathered the complete idea of how the constructor works and how to initialize the data members of the aggregated class.

Now the order of constructors and destructors is also important and often helps us to understand various concepts. To gather the idea about it check out this code,

```

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

class Triangle{
    Point p1, p2, p3;
public:
    Triangle():p1(1,1),p2(2,2),p3(3,3){
        cout << "Triangle Class Constructor\n";
    }
    ~Triangle(){
        cout << "Triangle Class Destructor\n";
    }
};

int main(){
    Triangle t;
    cout << "\nThese Lines are from main() function\n\n";
    return 0;
}

```

E:\program.exe

```

1 Point Class Constructor
2 Point Class Constructor
3 Point Class Constructor
Triangle Class Constructor

```

These Lines are from main() function

```

Triangle Class Destructor
3 Point Class Destructor
2 Point Class Destructor
1 Point Class Destructor

```

Process exited after 0.08088 seconds with return code 0
Press any key to continue . . .

Now, we are going to overload the stream operator (<<) to print our objects more precisely.

```

class Point{
    int x, y;
public:
    Point(int x, int y){
        this->x = x;
        this->y = y;
    }
    friend ostream& operator << (ostream &out, const Point &p)
    {
        out << '(' << p.x << ',' << p.y << ')' << '\n';
        return out;
    }
};

class Triangle{
    Point p1, p2, p3;
public:
    Triangle(int n[6])
        : p1(n[0], n[1]), p2(n[2], n[3]), p3(n[4], n[5]) { }
    friend ostream& operator << (ostream &out, const Triangle &t)
    {
        out << "Coordinates of Triangle are : \n";
        out << t.p1 << t.p2 << t.p3 << '\n';
        return out;
    }
};

int main(){
    int a[]={2, 5, 3, 4, -5, 2};
    Triangle t(a);
    cout << t;
    return 0;
}

```

E:\program.exe

```

Coordinates of Triangle are :
(2,5)
(3,4)
(-5,2)

```

Process exited after 0.07742 seconds with return code 0
Press any key to continue . . .

Here, again mentioning that we are only interested in the public members of the aggregated classes. As the stream operator is defined publicly in the Point class, we directly called the operator from our Triangle class to print the values of x and y coordinates. And the way we initialized the points in this program is also important. We can initialize points in many other ways as well, it all depends on the logic of the program.

Now, we are going to apply a practical example of aggregation step-by-step. We are using the relationship, “Employee has Address”.

1. Create an Address class with appropriate data members to represent a complete address.
2. Create a Constructor to initialize the Data members of Address Class (or you can also use setter functions to initialize members)
3. Overload stream operator (<<) to get formatted output of our Address. (or you can also use getters to return values and print them in main() or anywhere else). But, overloading (<<) operator is a better option.

Implementation of the above mentioned steps till now is as follows,

```
class Address  
{
```

```
    int houseNo;  
    char blockNo;  
    char town[30];  
    char city[30];
```

public:

```
    Address(int h, int b, char *t, char *c) // Parameterized
    Constructor
    {
        houseNo = h;
        blockNo = b;

        strcpy(town, t); //strcpy( ) is a built - in function used
        to copy one string (character array) to another, However, we can also
        define our own function.
        strcpy(city, c);
    }

    friend ostream& operator << (ostream &out, const Address &a)
    {
        out << "House No:" << a.houseNo << '\n';
        out << "Block No:" << a.blockNo << '\n';
        out << "Town:" << a.town << '\n';
        out << "City:" << a.city << '\n';
        return out;
    }
};
```

Next, we need to create an Employee class to save the data of Employees. An employee has the following attributes, employeeID, name, address, salary. But, here, for the purpose of simplification we only deal with the address of the employee to stay in the domain of aggregation.

1. Create a parameterized Constructor which holds the contents of the address of the employee.
2. Overload stream operator (<<) to print the data of an employee.

The Code for the Employee class holding these attributes will be as follows,

```

class Employee
{
    static int employeeCount; // To Assign employeeID
    int employeeID;
    char name[50];
    Address address;
    int salary;

public:
    // Parameterized Constructor to initialize the Address of the
    // Employee
    Employee(int s, char *n, int h, char b, char *t, char *c)
    : address(h,b,t,c)
    {
        employeeID = employeeCount;
        strcpy(n, name);
        salary = s;
        employeeCount++
    }

    friend ostream& operator << (ostream &out, const Employee &a)
    {
        out << "Employee No:" << a.employeeNo << '\n';
        out << "Name :" << a.name << '\n';
        out << "Salary:" << a.salary << '\n';
        out << "Address\n" << a.address; // Calls the stream
                                           operator from Address class
        return out;
    }
};

int Employee :: employeeCount = 1;

```


Now Writing the main() function of the program to test our above code for both of the classes.

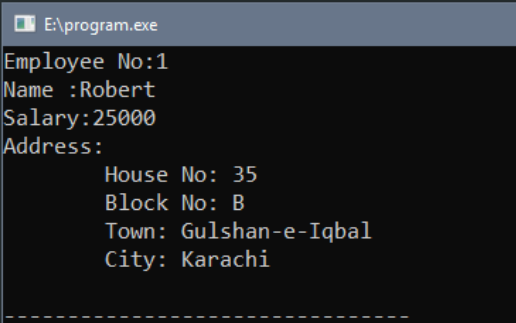
```
{
    static int employeeCount; // To Assign employeeID
    int employeeID;
    char name[50];
    Address address;
    int salary;

public:
    // Parameterized Constructor to initialize the Address of the Employee
    Employee(int s, char *n, int h, char b, char *t, char *c) : address(h,b,t,c)
    {
        employeeID = employeeCount;
        strcpy(name, n);
        salary = s;
        employeeCount++;
    }

    friend ostream& operator << (ostream &out, const Employee &a)
    {
        out << "Employee No:" << a.employeeID << '\n';
        out << "Name :" << a.name << '\n';
        out << "Salary:" << a.salary << '\n';
        out << "Address: \n" << a.address; // Calls the stream operator from Address class
        return out;
    }
};

int Employee :: employeeCount = 1;

int main()
{
    char name[] = "Robert";
    char town[] = "Gulshan-e-Iqbal";
    char city[] = "Karachi";
    Employee e(25000, name, 35, 'B', town, city );
    cout << e;
    return 0;
}
```



```
E:\program.exe
Employee No:1
Name :Robert
Salary:25000
Address:
    House No: 35
    Block No: B
    Town: Gulshan-e-Iqbal
    City: Karachi
-----
```

We can also make multiple objects in the same class. Like, we know that there may be two addresses stored for an employee : Current Address, and Permanent Address. If we want to store both of these addresses we can add 1 more data member of the Address class in the Employee class and store the 2nd address in that member.

And also we will create two Employees to show more data in our output.

Changing in Employee class to hold 2 addresses:

```
class Employee
{
    static int employeeCount; // To Assign employeeID
    int employeeID;
    char name[50];
    Address currentAddress;
    Address permanentAddress;
    int salary;

public:
    // Parameterized Constructor to initialize the Address of the Employee
    Employee(int s, char *n, int h, char b, char *t, char *c, int ph, char pb, char *pt, char *pc )
        : currentAddress(h,b,t,c) , permanentAddress(ph,pb,pt,pc)
    {
        employeeID = employeeCount;
        strcpy(name, n);
        salary = s;
        employeeCount++;
    }

    friend ostream& operator << (ostream &out, const Employee &a)
    {
        out << "Employee No:" << a.employeeID << '\n';
        out << "Name :" << a.name << '\n';
        out << "Salary:" << a.salary << '\n';
        out << "Current Address: \n" << a.currentAddress; // Calls the stream operator from Address class
        out << "Permanent Address: \n" << a.permanentAddress;
        return out;
    }
};

int Employee :: employeeCount = 1;
```

Output:

```
Employee No:1
Name :Robert
Salary:25000
Current Address:
    House No: 35
    Block No: B
    Town: Gulshan-e-Iqbal
    City: Karachi
Permanent Address:
    House No: 53
    Block No: J
    Town: Faisal Town
    City: Lahore
-----
Employee No:2
Name :Alan
Salary:55000
Current Address:
    House No: 15
    Block No: A
    Town: Bahria Twon
    City: Karachi
Permanent Address:
    House No: 21
    Block No: F
    Town: Gulberg
    City: Lahore
-----
```

-THE END-