16.2 — Composition

ALEX JULY 28, 2021

Object composition

In real-life, complex objects are often built from smaller, simpler objects. For example, a car is built using a metal frame, an engine, some tires, a transmission, a steering wheel, and a large number of other parts. A personal computer is built from a CPU, a motherboard, some memory, etc… Even you are built from smaller parts: you have a head, a body, some legs, arms, and so on. This process of building complex objects from simpler ones is called object composition.

Broadly speaking, object composition models a "has-a" relationship between two objects. A car "has-a" transmission. Your computer "has-a" CPU. You "have-a" heart. The complex object is sometimes called the whole, or the parent. The simpler object is often called the part, child, or component.

In C++, you've already seen that structs and classes can have data members of various types (such as fundamental types or other classes). When we build classes with data members, we're essentially constructing a complex object from simpler parts, which is object composition. For this reason, structs and classes are sometimes referred to as composite types.

Object Composition is useful in a C++ context because it allows us to create complex classes by combining simpler, more easily manageable parts. This reduces complexity, and allows us to write code faster and with less errors because we can reuse code that has already been written, tested, and verified as working.

Types of object composition

There are two basic subtypes of object composition: composition and aggregation. We'll examine composition in this lesson, and aggregation in the next.

A note on terminology: the term "composition" is often used to refer to both composition and aggregation, not just to the composition subtype. In this tutorial, we'll use the term "object composition" when we're referring to both, and "composition" when we're referring specifically to the composition subtype.

Composition

To qualify as a composition, an object and a part must have the following relationship:

- The part (member) is part of the object (class)
- The part (member) can only belong to one object (class) at a time
- The part (member) has its existence managed by the object (class)
- The part (member) does not know about the existence of the object (class)

A good real-life example of a composition is the relationship between a person's body and a heart. Let's examine these in more detail.

Composition relationships are part-whole relationships where the part must constitute part of the whole object. For example, a heart is a part of a person's body. The part in a composition can only be part of one object at a time. A heart that is part of one person's body can not be part of someone else's body at the same time.

In a composition relationship, the object is responsible for the existence of the parts. Most often, this means the part is created when the object is created, and destroyed when the object is destroyed. But more broadly, it means the object manages the part's lifetime in such a way that the user of the object does not need to get involved. For example, when a body is created, the heart is created too. When a person's body is destroyed, their heart is destroyed too. Because of this, composition is sometimes called a "death relationship".

And finally, the part doesn't know about the existence of the whole. Your heart operates blissfully unaware that it is part of a larger structure. We call this a unidirectional relationship, because the body knows about the heart, but not the other way around.

Note that composition has nothing to say about the transferability of parts. A heart can be transplanted from one body to another. However, even after being transplanted, it still meets the requirements for a composition (the heart is now owned by the recipient, and can only be part of the recipient object unless transferred again).

Our ubiquitous Fraction class is a great example of a composition:

```
class Fraction
{
private:
 int m_numerator;
 int m_denominator;

public:
 Fraction(int numerator=0, int denominator=1)
  : m_numerator{ numerator }, m_denominator{ denominator }
 {
  // We put reduce() in the constructor to ensure any fractions we make get reduced!
  // Since all of the overloaded operators create new Fractions, we can guarantee this will get called here
  reduce();
 }
};
```

This class has two data members: a numerator and a denominator. The numerator and denominator are part of the Fraction (contained within it). They can not belong to more than one Fraction at a time. The numerator and denominator don't know they are part of a Fraction, they just hold integers. When a Fraction instance is created, the numerator and denominator are created. When the fraction instance is destroyed, the numerator and denominator are destroyed as well.

While object composition models has-a type relationships (a body has-a heart, a fraction has-a denominator), we can be more precise and say that composition models "part-of" relationships (a heart is part-of a body, a numerator is part of a fraction). Composition is often used to model physical relationships, where one object is physically contained inside another.

The parts of a composition can be singular or multiplicative -- for example, a heart is a singular part of the body, but a body contains 10 fingers (which could be modeled as an array).

Implementing compositions

Compositions are one of the easiest relationship types to implement in C++. They are typically created as structs or classes with normal data members. Because these data members exist directly as part of the struct/class, their lifetimes are bound to that of the class instance itself.

Compositions that need to do dynamic allocation or deallocation may be implemented using pointer data members. In this case, the composition class should be responsible for doing all necessary memory management itself (not the user of the class).

In general, if you *can* design a class using composition, you *should* design a class using composition. Classes designed using composition are straightforward, flexible, and robust (in that they clean up after themselves nicely).

More examples

Many games and simulations have creatures or objects that move around a board, map, or screen. One thing that all of these creatures/objects have in common is that they all have a location. In this example, we are going to create a creature class that uses a point class to hold the creature's location.

First, let's design the point class. Our creature is going to live in a 2d world, so our point class will have 2 dimensions, X and Y. We will assume the world is made up of discrete squares, so these dimensions will always be integers.

Point2D.h:

```cpp
#ifndef POINT2D_H
#define POINT2D_H

#include <iostream>

class Point2D
{
private:
    int m_x;
    int m_y;

public:
    // A default constructor
    Point2D()
        : m_x{ 0 }, m_y{ 0 }
    {
    }

    // A specific constructor
    Point2D(int x, int y)
        : m_x{ x }, m_y{ y }
    {
    }

    // An overloaded output operator
    friend std::ostream& operator<<(std::ostream& out, const Point2D& point)
    {
        out << '(' << point.m_x << ", " << point.m_y << ')';
        return out;
    }

    // Access functions
    void setPoint(int x, int y)
    {
        m_x = x;
        m_y = y;
    }

};

#endif
```

Note that because we've implemented all of our functions in the header file (for the sake of keeping the example concise), there is no Point2D.cpp.

This Point2d class is a composition of its parts: location values x and y are part-of Point2D, and their lifespan is tied to that of a given Point2D instance.

Now let's design our Creature. Our Creature is going to have a few properties: a name, which will be a string, and a location, which will be our Point2D class.

Creature.h:

```
#ifndef CREATURE_H
#define CREATURE_H

#include <iostream>
#include <string>
#include "Point2D.h"

class Creature
{
private:
    std::string m_name;
    Point2D m_location;

public:
    Creature(const std::string& name, const Point2D& location)
        : m_name{ name }, m_location{ location }
    {
    }

    friend std::ostream& operator<<(std::ostream& out, const Creature&
creature)
    {
        out << creature.m_name << " is at " << creature.m_location;
        return out;
    }

    void moveTo(int x, int y)
    {
        m_location.setPoint(x, y);
    }
};
#endif
```

This Creature is also a composition of its parts. The creature's name and location have one parent, and their lifetime is tied to that of the Creature they are part of.

And finally, Main.cpp:

```cpp
#include <string>
#include <iostream>
#include "Creature.h"
#include "Point2D.h"

int main()
{
    std::cout << "Enter a name for your creature: ";
    std::string name;
    std::cin >> name;
    Creature creature{ name, { 4, 7 } };

    while (true)
    {
        // print the creature's name and location
        std::cout << creature << '\n';

        std::cout << "Enter new X location for creature (-1 to quit): ";
        int x{ 0 };
        std::cin >> x;
        if (x == -1)
            break;

        std::cout << "Enter new Y location for creature (-1 to quit): ";
        int y{ 0 };
        std::cin >> y;
        if (y == -1)
            break;

        creature.moveTo(x, y);
    }

    return 0;
}
```

Here's a transcript of this code being run:

```
Enter a name for your creature: Marvin
Marvin is at (4, 7)
Enter new X location for creature (-1 to quit): 6
Enter new Y location for creature (-1 to quit): 12
Marvin is at (6, 12)
Enter new X location for creature (-1 to quit): 3
Enter new Y location for creature (-1 to quit): 2
Marvin is at (3, 2)
Enter new X location for creature (-1 to quit): -1
```

Variants on the composition theme

Although most compositions directly create their parts when the composition is created and directly destroy their parts when the composition is destroyed, there are some variations of composition that bend these rules a bit.

For example:

- A composition may defer creation of some parts until they are needed. For example, a string class may not create a dynamic array of characters until the user assigns the string some data to hold.
- A composition may opt to use a part that has been given to it as input rather than create the part itself.
- A composition may delegate destruction of its parts to some other object (e.g. to a garbage collection routine).

The key point here is that the composition should manage its parts without the user of the composition needing to manage anything.

Composition and subclasses

One question that new programmers often ask when it comes to object composition is, "When should I use a subclass instead of direct implementation of a feature?". For example, instead of using the Point2D class to implement the Creature's location, we could have instead just added 2 integers to the Creature class and written code in the Creature class to handle the positioning. However, making Point2D its own class has a number of benefits:

1. Each individual class can be kept relatively simple and straightforward, focused on performing one task well. This makes those classes easier to write and much easier to understand, as they are more focused. For example, Point2D only worries about point-related stuff, which helps keep it simple.
2. Each subclass can be self-contained, which makes them reusable. For example, we could reuse our Point2D class in a completely different application. Or if our creature ever needed another point (for example, a destination it was trying to get to), we can simply add another Point2D member variable.
3. The parent class can have the subclasses do most of the hard work, and instead focus on coordinating the data flow between the subclasses. This helps lower the overall complexity of the parent object, because it can delegate tasks to its children, who already know how to do those tasks. For example, when we move our Creature, it delegates that task to the Point class, which already understands how to set a point. Thus, the Creature class does not have to worry about how such things would be implemented.

Tip

A good rule of thumb is that each class should be built to accomplish a single task. That task should either be the storage and manipulation of some kind of data (e.g. Point2D, std::string), OR the coordination of subclasses (e.g. Creature). Ideally not both.

In this case of our example, it makes sense that Creature shouldn't have to worry about how Points are implemented, or how the name is being stored. Creature's job isn't to know those intimate details. Creature's job is to worry about how to coordinate the data flow and ensure that each of the subclasses knows *what* it is supposed to do. It's up to the individual subclasses to worry about *how* they will do it.

Next lesson
16.3 Aggregation

Back to table of contents

Previous lesson
16.1 Object relationships

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest

X

X