16.3 — Aggregation

ALEX SEPTEMBER 1, 2021

In the previous lesson 16.2 -- Composition, we noted that object composition is the process of creating complex objects from simpler ones. We also talked about one type of object composition, called composition. In a composition relationship, the whole object is responsible for the existence of the part.

In this lesson, we'll take a look at the other subtype of object composition, called aggregation.

Aggregation

To qualify as an aggregation, a whole object and its parts must have the following relationship:

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

Like a composition, an aggregation is still a part-whole relationship, where the parts are contained within the whole, and it is a unidirectional relationship. However, unlike a composition, parts can belong to more than one object at a time, and the whole object is not responsible for the existence and lifespan of the parts. When an aggregation is created, the aggregation is not responsible for creating the parts. When an aggregation is destroyed, the aggregation is not responsible for destroying the parts.

For example, consider the relationship between a person and their home address. In this example, for simplicity, we'll say every person has an address. However, that address can belong to more than one person at a time: for example, to both you and your roommate or significant other. However, that address isn't managed by the person -- the address probably existed before the person got there, and will exist after the person is gone. Additionally, a person knows what address they live at, but the addresses don't know what people live there. Therefore, this is an aggregate relationship.

Alternatively, consider a car and an engine. A car engine is part of the car. And although the engine belongs to the car, it can belong to other things as well, like the person who owns the car. The car is not responsible for the creation or destruction of the engine. And while the car knows it has an engine (it has to in order to get anywhere) the engine doesn't know it's part of the car.

When it comes to modeling physical objects, the use of the term "destroyed" can be a little dicey. One might argue, "If a meteor fell out of the sky and crushed the car, wouldn't the car parts all be destroyed too?" Yes, of course. But that's the fault of the meteor. The important point is that the car is not responsible for destruction of its parts (but an external force might be).

We can say that aggregation models "has-a" relationships (a department has teachers, the car has an engine).

Similar to a composition, the parts of an aggregation can be singular or multiplicative.

Implementing aggregations

Because aggregations are similar to compositions in that they are both part-whole relationships, they are implemented almost identically, and the difference between them is mostly semantic. In a composition, we typically add our parts to the composition using normal member variables (or pointers where the allocation and deallocation process is handled by the composition class).

In an aggregation, we also add parts as member variables. However, these member variables are typically either references or pointers that are used to point at objects that have been created outside the scope of the class. Consequently, an aggregation usually either takes the objects it is going to point to as constructor parameters, or it begins empty and the subobjects are added later via access functions or operators.

Because these parts exist outside of the scope of the class, when the class is destroyed, the pointer or reference member variable will be destroyed (but not deleted). Consequently, the parts themselves will still exist.

Let's take a look at a Teacher and Department example in more detail. In this example, we're going to make a couple of simplifications: First, the department will only hold one teacher. Second, the teacher will be unaware of what department they're part of.

```
#include <iostream>
#include <string>

class Teacher
{
private:
  std::string m_name{};

public:
  Teacher(const std::string& name)
      : m_name{ name }
  {
  }

  const std::string& getName() const { return m_name; }
};

class Department
{
private:
  const Teacher& m_teacher; // This dept holds only one teacher for simplicity, but it could hold many teachers

public:
  Department(const Teacher& teacher)
      : m_teacher{ teacher }
  {
  }
};

int main()
{
  // Create a teacher outside the scope of the Department
  Teacher bob{ "Bob" }; // create a teacher

  {
    // Create a department and use the constructor parameter to pass
    // the teacher to it.
    Department department{ bob };

  } // department goes out of scope here and is destroyed

  // bob still exists here, but the department doesn't

  std::cout << bob.getName() << " still exists!\n";

  return 0;
}
```

In this case, `bob` is created independently of `department`, and then passed into `department`'s constructor. When `department` is destroyed, the `m_teacher` reference is destroyed, but the teacher itself is not destroyed, so it still exists until it is independently destroyed later in `main()`.

Pick the right relationship for what you're modeling

Although it might seem a little silly in the above example that the Teachers don't know what Department they're working for, that may be totally fine in the context of a given program. When you're determining what kind of relationship to implement, implement the simplest relationship that meets your needs, not the one that seems like it would fit best in a real-life context.

For example, if you're writing a body shop simulator, you may want to implement a car and engine as an aggregation, so the engine can be removed and put on a shelf somewhere for later. However, if you're writing a racing simulation, you may want to implement a car and an engine as a composition, since the engine will never exist outside of the car in that context.

Best practice

Implement the simplest relationship type that meets the needs of your program, not what seems right in real-life.

Summarizing composition and aggregation

Compositions:

- Typically use normal member variables
- Can use pointer members if the class handles object allocation/deallocation itself
- Responsible for creation/destruction of parts

Aggregations:

- Typically use pointer or reference members that point to or reference objects that live outside the scope of the aggregate class
- Not responsible for creating/destroying parts

It is worth noting that the concepts of composition and aggregation can be mixed freely within the same class. It is entirely possible to write a class that is responsible for the creation/destruction of some parts but not others. For example, our Department class could have a name and a Teacher. The name would probably be added to the Department by composition, and would be created and destroyed with the Department. On the other hand, the Teacher would be added to the department by aggregation, and created/destroyed independently.

While aggregations can be extremely useful, they are also potentially more dangerous, because aggregations do not handle deallocation of their parts. Deallocations are left to an external party to do. If the external party no longer has a pointer or reference to the abandoned parts, or if it simply forgets to do the cleanup (assuming the class will handle that), then memory will be leaked.

For this reason, compositions should be favored over aggregations.

A few warnings/errata

For a variety of historical and contextual reasons, unlike a composition, the definition of an aggregation is not precise -- so you may see other reference material define it differently from the way we do. That's fine, just be aware.

One final note: In the lesson Structs, we defined aggregate data types (such as structs and classes) as data types that group multiple variables together. You may also run across the term aggregate class in your C++ journeys, which is defined as a struct or class that has no provided constructors, destructors, or overloaded assignment, has all public members, and does not use inheritance -- essentially a plain-old-data struct. Despite the similarities in naming, aggregates and aggregation are different and should not be confused.

---

std::reference_wrapper

In the `Department` / `Teacher` example above, we used a reference in the `Department` to store the `Teacher`. This works fine if there is only one `Teacher`, but what if a Department has multiple Teachers? We'd like to store those Teachers in a list of some kind (e.g. a `std::vector`) but fixed arrays and the various the standard library lists can't hold references (because list elements must be assignable, and references can't be reassigned).

```
std::vector<const Teacher&> m_teachers{}; // Illegal
```

Instead of references, we could use pointers, but that would open the possibility to store or pass null pointers. In the `Department` / `Teacher` example, we don't want to allow null pointers. To solve this, there's `std::reference_wrapper`.

Essentially, `std::reference_wrapper` is a class that acts like a reference, but also allows assignment and copying, so it's compatible with lists like `std::vector`.

The good news is that you don't really need to understand how it works to use it. All you need to know are three things:

1. **`std::reference_wrapper` lives in the <functional> header.**
2. **When you create your `std::reference_wrapper` wrapped object, the object can't be an anonymous object (since anonymous objects have expression scope, and this would leave the reference dangling).**
3. **When you want to get your object back out of `std::reference_wrapper`, you use the `get()` member function.**

Here's an example using `std::reference_wrapper` in a `std::vector`:

```
#include <functional> // std::reference_wrapper
#include <iostream>
#include <vector>
#include <string>

int main()
{
  std::string tom{ "Tom" };
  std::string berta{ "Berta" };

  std::vector<std::reference_wrapper<std::string>> names{ tom, berta }; // these strings are stored by reference, not value

  std::string jim{ "Jim" };

  names.push_back(jim);

  for (auto name : names)
  {
    // Use the get() member function to get the referenced string.
    name.get() += " Beam";
  }

  std::cout << jim << '\n'; // Jim Beam

  return 0;
}
```

To create a vector of const references, we'd have to add const before the `std::string` like so

```
// Vector of const references to std::string
std::vector<std::reference_wrapper<const std::string>> names{ tom, berta
};
```

If this seems a bit obtuse or obscure at this point (especially the nested types), come back to it later after we've covered template classes and you'll likely find it more understandable.

Quiz time

Question #1

Would you be more likely to implement the following as a composition or an aggregation?
a) A ball that has a color
b) An employer that is employing multiple people
c) The departments in a university
d) Your age
e) A bag of marbles

Show Solution

Question #2

Update the `Department` / `Teacher` example so the `Department` can handle multiple Teachers. The following code should execute:

```
#include <iostream>

// ...

int main()
{
  // Create a teacher outside the scope of the Department
  Teacher t1{ "Bob" };
  Teacher t2{ "Frank" };
  Teacher t3{ "Beth" };

  {
    // Create a department and add some Teachers to it
    Department department{}; // create an empty Department

    department.add(t1);
    department.add(t2);
    department.add(t3);

    std::cout << department;

  } // department goes out of scope here and is destroyed

  std::cout << t1.getName() << " still exists!\n";
  std::cout << t2.getName() << " still exists!\n";
  std::cout << t3.getName() << " still exists!\n";

  return 0;
}
```

This should print:

```
Department: Bob Frank Beth
Bob still exists!
Frank still exists!
Beth still exists!
```

Show Hint

Show Solution

Next lesson
16.4 Association

Back to table of contents

Previous lesson
16.2 Composition

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