10.2 — Composition

BY ALEX ON DECEMBER 4TH, 2007 | LAST MODIFIED BY ALEX ON FEBRUARY 28TH, 2016

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 **composition** (also known as object composition).

More specifically, composition is used for objects that have a *has-a* relationship to each other. A car *has-a* metal frame, *has-an* engine, and *has-a* transmission. A personal computer *has-a* CPU, a motherboard, and other components. You *have-a* head, a body, some limbs.

So far, all of the classes we have used in our examples have had member variables that are built-in data types (eg. int, double). While this is generally sufficient for designing and implementing small, simple classes, it quickly becomes burdensome for more complex classes, especially those built from many sub-parts. In order to facilitate the building of complex classes from simpler ones, C++ allows us to do object composition in a very simple way -- by using classes as member variables in other classes.

Lets take a look at some examples of how this is done. If we were designing a personal computer class, we might do it like this (assuming we'd already written a CPU, Motherboard, and RAM class):

```
#include "CPU.h"
#include "Motherboard.h"
#include "RAM.h"

class PersonalComputer
{
private:
    CPU m_cCPU;
    Motherboard m_cMotherboard;
    RAM m_cRAM;
};
```

Initializing class member variables

In the previous lesson on **initializer lists**, you learned that the preferred way to initialize class members is through initializer lists rather than assignment. So let's write a constructor for our PersonalComputer class that uses an initialization list to initialize the member variables. This constructor will take 3 parameters: a CPU speed, a motherboard model, and a RAM size, which it will then pass to the respective member variables when they are constructed.

```
PersonalComputer::PersonalComputer(int nCPUSpeed, char *strMotherboardModel, int nRAMSize)
    : m_cCPU(nCPUSpeed), m_cMotherboard(strMotherboardModel), m_cRAM(nRAMSize)
{
}
```

Now, when a PersonalComputer object is instantiated using this constructor, that PersonalComputer object will contain a CPU object initialized with nCPUSpeed, a Motherboard object initialized with strMotherboardModel, and a RAM object initialized with nRAMSize.

It is worth explicitly noting that composition implies ownership between the complex class and any subclasses. When the complex class is created, the subclasses are created. When the complex class is destroyed, the subclasses are similarly destroyed.

A full example

While the above example is useful in giving the general idea of how composition works, let's do a full example that you can compile yourself. Many games and simulations have creatures or objects that move around a board, map, or screen. The 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:

```
#ifndef POINT2D_H
#define POINT2D_H

#include <iostream>

class Point2D
{
private:
    int m_nX;
    int m_nY;

public:
    // A default constructor
    Point2D()
        : m_nX(0), m_nY(0)
    {
    }

    // A specific constructor
    Point2D(int nX, int nY)
        : m_nX(nX), m_nY(nY)
    {
    }

    // An overloaded output operator
    friend std::ostream& operator<<(std::ostream& out, const Point2D &cPoint)
    {
        out << "(" << cPoint.GetX() << ", " << cPoint.GetY() << ")";
        return out;
    }

    // Access functions
    void SetPoint(int nX, int nY)
    {
        m_nX = nX;
        m_nY = nY;
    }

    int GetX() const { return m_nX; }
    int GetY() const { return m_nY; }
};

#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.

Now let's design our Creature. Our Creature is going to have a few properties. It's going to have 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"
```

```
7
8   class Creature
9   {
10  private:
11      std::string m_strName;
12      Point2D m_cLocation;
13
14  public:
15      Creature(std::string strName, const Point2D &cLocation)
16          : m_strName(strName), m_cLocation(cLocation)
17      {
18      }
19
20      friend std::ostream& operator<<(std::ostream& out, const Creature &cCreature)
21      {
22          out << cCreature.m_strName << " is at " << cCreature.m_cLocation;
23          return out;
24      }
25
26      void MoveTo(int nX, int nY)
27      {
28          m_cLocation.SetPoint(nX, nY);
29      }
30  };
31  #endif
```

And finally, Main.cpp:

```
1   #include <string>
2   #include <iostream>
3   #include "Creature.h"
4   #include "Point2D.h"
5
6   int main()
7   {
8       using namespace std;
9       cout << "Enter a name for your creature: ";
10      std::string cName;
11      cin >> cName;
12      Creature cCreature(cName, Point2D(4, 7));
13
14      while (1)
15      {
16          cout << cCreature << endl;
17          cout << "Enter new X location for creature (-1 to quit): ";
18          int nX=0;
19          cin >> nX;
20          if (nX == -1)
21              break;
22
23          cout << "Enter new Y location for creature (-1 to quit): ";
24          int nY=0;
25          cin >> nY;
26          if (nY == -1)
27              break;
28
29          cCreature.MoveTo(nX, nY);
30          }
31
32      return 0;
33  }
```

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
```

Why use composition?

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, using composition provides a number of useful benefits:

1. Each individual class can be kept relatively simple and straightforward, focused on performing one task. This makes those classes easier to write and much easier to understand. For example, Point2D only worries about point-related stuff, which helps keep it simple.
2. Each subobject 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 complex class can have the simple 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 complex object, because it can delegate tasks to the sub-objects, who already know how to do them. 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.

One question that new programmers often ask is "When should I use composition instead of direct implementation of a feature?". There's no 100% answer to that question. However, 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 (eg. Point2D), OR the coordination of subclasses (eg. Creature). 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.


10.3 -- Aggregation


Index


9.12 -- Shallow vs deep copying

Share this: