CS 246 Fall 2016 - Tutorial 10

November 16, 2016

1 Summary

- Forward Declarations
- Map
- Template Method Pattern
- Factory Pattern
- Visitor Pattern

2 Tip of the Week:

-

3 Forward Declarations

- Consider the following code.

```
class Person{
  vector<Cat*> cats;
  std::string name;
 public:
  void adopt(Cat*)
  Person(std::string);
};

class Cat{
  Person* minion;
  std::string name;
 public:
  void addMinion(Person*);
  Cat(std::string);
};
```

- These definitions are going to be separated into two header files. We notice that Person needs to know about Cat because the class has a vector of Cat pointers and Cat needs to know about Person because it has a Person pointer. We will include the header file for each in the header file for the other one.
- This causes a compilation error. Why? Consider person.h. It includes cat.h which means the contents of cat.h will appear at the top of person.h. We have cat.h including person.h, but due to our include guard, the contents of person.h is not place here. Now we have the definition of Cat occuring before Person in this file and the Cat has no idea what a person is.
- Think about the definition of Person. Does it realy need to know everything about Cat? What does it need to know about Cat? It really just needs to know that the class Cat exists. Instead of including the Cat definition, all we need to do it forward declare Cat. `class Cat;` This is similar to declaring a function befor defining it.

- In general, when is it appropriate to forward declare a class and when do we need to have the class definition?

  Any time we need to know anything specific about a class, we need to have the class definition. This include if we need to know the size, fields or methods of the class.

  Any time we just need to know that the class exists, we can forward declare the class. If the only times we are using the class we are using a pointer or a reference to the class, we just need to know if the class exists.

- Consider the code below. Which classes do we need to define and which ones can we forward declare?

```
struct K: public A{
  B b;
  C* c;
  D& d;
  void foo(E);
  void foo(F*);
  void foo(G&);
  H func();
  I* func2();
  J& func3();
};

std::istream& operator<<(std::istream&, const K&);
```

4 Map

- Maps are a container in the standard template library. Each entry in a map is a key-value pair. To create a map, you must give the types for both the key and value. i.e. `map<int,string> m;`
- To place an object in a map, `m.emplace(5,''hello'')`. Using emplace will create a copy of the key and the value to be stored in the map. The pair will only be inserted if the key does not currently exist in map.
- Alternatively, `m[9] = ''world'';` can be used for placing pairs. This will insert if 9 is not currently a key and overwrite the value if 9 currently is a key.
- Map iterators start at smallest key in the map and go through all keys in order.

5 Factory Pattern

- Problem: at run-time, we want to create instances of a subclass based on some criteria. Some examples of these criteria are input from the user, and a specified probability distribution. We also want to be able to easily change the criteria.
- Solution: create a class which has a method which creates instances of the subclass. If we desire being able to switch the criteria, the class can be abstract.
- Example: pizza factory.

6 Visitor Pattern

- Problem: at run-time, we want to decide the behaviour of the program based on the type of two classes (double dispatch) while ensuring our classes are not too highly coupled and easily expandable.
- Solution: create two abstract classes, one will the objects Visited and the other will the Visitor. The class which is visited will have a pure virtual method which accepts a Visitor reference, call it visit for now. The Visitor class will have an overloaded method where each instance takes in a different subtype of the Visited class, call it accept for now.

Each subclass of Visited, will implement its visit method to call accept with a this dereferenced. Each subclass of Visitor will implement accept for each subclass of Visited. Now when visit is called on a class, it will call the appropriate subclass's visit method. When we pass a dereferenced version of this to the accept method, it calls the method of Visitor which matches this' type.

We've now decided which method to call based on the type of two classes.

- Note: the Visitor class and its subclasses are highly coupled to the Visited class. Is this a problem?
- Example: tree.

7 Template Method Pattern

- Problem: we want to allow subclasses to have different implementations of some feacture while forcing the public interface of the classes and not allowing the the user/other programmers full access.
- Solution: Implement an abstract superclass which has public non-virtual method(s) and private virtual method(s). Have the superclass non-virtual method perform the operations which will be the same in all subclasses. For subclass implementation, call the virtual methods.
- Example: faces and turtles.