C++ Primer Plus, 5th *Edition* by Stephen Prata
Chapter 15: Friends, Exceptions, and More
Review Questions

1. What's wrong with the following attempts at establishing friends?

   a.
   ```
   class snap {
       friend clasp;
       ...
   };
   class clasp { ... };
   ```
   The line of code `friend clasp` *in the* `snap` *class should be* `friend class clasp` *so the compiler knows that* `clasp` *is a class.*

   b.
   ```
   class cuff {
   public:
       void snip(muff &) { ... }
       ...
   };
   class muff {
       friend void cuff::snip(muff &);
       ...
   };
   ```
   The `cuff` *class uses a* `muff` *reference in as a formal argument in a member function. However, the compiler has no way of knowing what type* `muff` *is. To solve this problem, we should insert a forward declaration before we define the* `cuff` *class that reads* `class muff;`

   c.
   ```
   class muff {
       friend void cuff::snip(muff &);
       ...
   };
   class cuff {
   public:
       void snip(muff &) { ... }
       ...
   };
   ```
   The complier needs to see the definition of the `cuff` *class before we can declare its functions friends to another class. We can't solve this by a forward declaration at the top since the definition of the* `cuff` *class must precede that of the* `muff` *class. However, this this suffers the same problem as part (b), which can be addresses by placing a forward declaration for the* `muff` *class at the top.*

2. You've seen how to create mutual class friends. Can you create a more restricted form of friendship in which only some members of Class B are friends to Class A and some members of A are friends to B? Explain.

   This is a tricky situation. The compiler requires that we place a class definition before specific functions from that class are declared as friends to another class. However, if we want to make both classes such that each class has functions that are friends of the other class, we clearly have a problem. A forward declaration is only helpful when we use a type that is of a class not declared yet; it is of no help if we wish to declare a class's functions as friends of another class before the original class is defined. So as we can see, we can only have one class have member functions which are friends to the other class. Otherwise, it is best to make two classes mutual friends.

3. What problems might the following nested class declaration have?

```
class Ribs
{
private:
    class Sauce
    {
        int soy;
        int sugar;
    public:
        Sauce(int s1, int s2) : soy(s1), sugar(s2) { }
    };
    ...
};
```

   The problem is that we have no way of accessing the variables `soy` *and* `sugar` *of the* `Sauce` *class once we construct an object of that class. The private variables of the* `Sauce` *class can only be accessed by the* `Sauce` *class directly and can only be accessed by the* `Ribs` *class indirectly through the public member functions of the* `Sauce` *class. In this case, there are none which return the values* `soy` *and* `sugar`*.*

4. How does `throw` differ from `return`?

   When throw is called, control is moved to successive calling functions until the first `try-catch` *block is reached which matches the exception type thrown, or else the program aborts because of an unexpected exception. When return is called control goes directly to the calling function. Both share the property that automatic variables declared after the function that the control is given to are deallocated from memory.*

5. Suppose you have a hierarchy of exception classes that are derived from a base exception class. In what order should you place `catch` blocks?

   `catch` *blocks should be placed in decending order of derivation. That is,* `catch` *blocks whose argument is a derived class should be placed before the* `catch` *block of a base class.*

6. Consider the `Grand`, `Superb`, and `Magnificent` classes defined in this chapter. Suppose `pg` is a type `Grand` * pointer that is assigned the address of an object of one of these three classes and that `ps` is a type `Superb` * pointer. What is the difference in how the following two code samples behave?

```
if (ps = dynamic_cast<Superb *>(pg))
    ps->say();    // sample #1

if (typeid(*pg) == typeid(Superb))
    (Superb *) pg->say();    // sample #2
```

   The first code sample will execute the statement `ps->say();` *if* `pg` *points to a Superb or Magnificent object. Since the* `say()` *function is virtual, the function call will correspond to the object* `pg` *points to. In the event that* `pg` *points to a* `Grand` *object, the test statement will evaluate as false and nothing will happen.*

   The second code sample will execute the `say()` *member function of the object* `pg` *points to only if* `pg` *points to a* `Superb` *object.*

7. How is the `static_cast` operator different from the `dynamic_cast` operator?

   The operators have the form
   `dummy_cast <` *`type-name`* `> (`*`expression`*`).`
   The `static_cast` *operator will make the conversion if the expression can be converted to the type specified by type-name implicitly or vica versa. The* `dynamic_cast` *operator will allow us to assign a pointer to an object only if the expression can be implicitly converted to the type-specified by type-name.*