CS 247 S2010 Midterm

1. A) The scope qualifier is used to identify identifiers used in different scopes, such as using member functions of a class outside that class.

B) We can make the copy constructor and copy assignment operator private.

C)
Invalid valid invalid invalid
Invalid invalid invalid invalid
Valid valid valid valid

2. A) int* const ptr = new int(3);

B) Not declaring o to be constant means the function could be written to accidentally modify o when it shouldn't be modified.

C) The implicit this parameter is the left-hand side of the operator.

D)
```
struct YearImpl {
        int year;
};
class Year {
        YearImpl *pImpl;
        public:
                Year(int);
                int year();
};
```

3. A)
```
#ifndef A_H
#define A_H

class B; // Forward declaration
…
```

B)
```
A.o : A.cpp, A.h, B.h
\t g++ -std=c++14 -c A.cpp
```

C) Placing the using directive in a header file forces all files importing that header to use that namespace.

D)
```
// Method 1
N::i
```

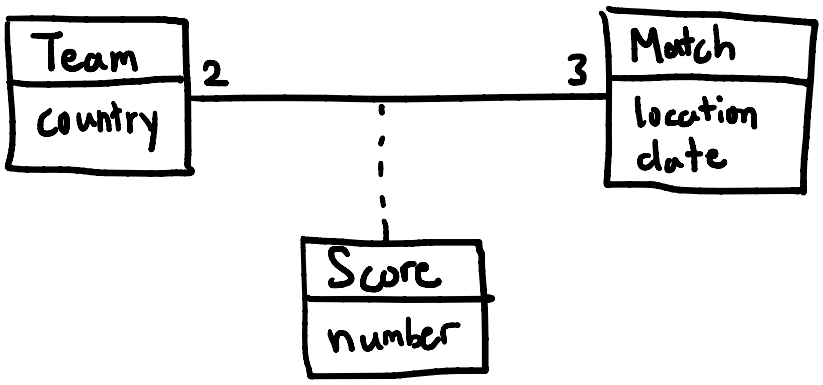
```
// Method 2
using namespace N;
i = 0;

// Method 3
???
```

4. A) i. It violates the law. afunc is calling bfunc1, then calling its cfunc - not immediate friend.
ii. It violates the law. Afunc is calling bfunc2, then calling its bfunc3.
iii. It does not violate the law.

B) We can make a mock of the unimplemented modules.

C) It is helpful for detecting errors that occur at boundary values of inputs, and finding anomalies.

D)



5. A)
i. Base::func()
ii. Base::vfunc()
iii. Derived::vfunc2()
iv. Base::vfunc()
v. Derived::vfunc3()

B)
func -> Base::func()
vfunc -> Base::vfunc()
vfunc2 -> Derived::vfunc2()
Vfunc3 -> Derived::vfunc3()

6. A)

```
class PostalCode {
        const std::string postalCode;
        bool isValid();
public:
        PostalCode(std::string postalCode);
```

```
    friend ostream& operator<<(ostream&, const PostalCode &p);
};
```

B) The postal code string is immutable, since the postal code of an address should not change.

C) There is no default initial value. An initialized postal code will always have a valid value.

D) isValid is a member function since it is specific to PostalCode and access private member variables. The ostream operator is not a member function, since the left-hand operator is always implicitly *this, and we want that to be std.

E) isValid checks the value of the postal code object is valid.

7.

A)

```
std::vector<Ingredient> ingredients;
```

B)

```
Recipe::printRecipe(std::iostream& out) {
    out << "Recipe Name: " << name << std::endl;
    out << "Ingredients:" << std::endl;
    for (const auto &ingredient: ingredients) {
        ingredient.print(out);
        out << std::endl;
    }
    out << "Instructions:" << std::endl << instructions << std::endl;
}
```

8.

```
class Food {
    std::string name;
protected:
    Food(std::string name);
public:
    ~Food() = 0;
    void printName(std::iostream&) final;
    virtual void print(std::iostream&);
};

Food::Food(std::string name) : name(name) {}
Food::~Food() {}
void Food::printName(std::iostream& out) final {
    out << name;
}
void Food::print(std::iostream& out) {
    printName(out);
}
```

9. A)

```
class Ingredient : public Food {
        float amount;
protected:
        Unit unit;
public:
        Ingredient(std::string name, float amount, unit = "");
        float amount();
        void amountIs(float);
        void print(std::iostream&);
};
```

B)

```
void Ingredient::print(std::iostream& out) {
        out << amount << " ";
        unit.print(out);
        printName(out);
}
```

C)

```
void Prepared_Ingredient::print(std::iostream& out) {
        out << amount << " ";
        unit.print(out);
        printName(out);
        out << ", " << preparation;
}
```

10. A)

```
Ingredient::Ingredient(std::string name, float amount, std::string unit) : Food(name),
amount(amount), unit(Unit{unit}) {}
```

B)

```
Ingredient& Ingredient::operator= (Ingredient& other) {
        Ingredient temp{other};
        using std::swap;
        swap(amount, temp.amount);
        swap(unit, temp.unit);
        return *this;
}
```

11. Copy constructor

- The copy constructor initializes a new PreparedIngredient object with another PreparedIngredient object.
- A new one should be created, since Ingredients contain other objects that need to be properly deep copied, like units.

Copy assignment operator

- The copy assignment operator assigns an already initialized PreparedIngredient object to a copy of another PreparedIngredient object.
- A new one should be created, since Ingredients contain other objects that need to be properly deep copied, like units.

Destructor

- Cleanup of the PreparedIngredient object before it is deleted/goes out of scope.
- If the unit is a pointer on the heap, a destructor needs to be created to free that memory.

12. A) The Liskov Substitutability Principle states that pointers to a Base class must be blindly substitutable by child class objects. Since Ingredient is a subclass of Food, an Ingredient object should be substitutable for a Food object.

B) In this instance, an Ingredient is not a more specific Food - it contains Food but also has other data members and functions that are not general Food.


```
class Ingredient {
        float amount;
protected:
        Unit unit;
        Food food;
public:
        Ingredient(std::string name, float amount, unit = "");
        float amount();
        void amountIs(float);
        void print(std::iostream&);
};
```