Exercise 1 ....

(a) You will first build two classes, `Mammal` and `Dog`. `Dog` will inherit from `Mammal`. Below is the `Mammal` class code. Once you have the `Mammal` class built, build a second class `Dog` that will inherit publicly from `Mammal`.

```
class Mammal {
    public:
    Mammal(void) : itsAge(1) {
      cout << "Mammal constructor..." << endl;
    }
    ~Mammal(void) {
      cout << "Mammal destructor..." << endl;
    }
    virtual void Move() const {
        cout << "Mammal moves a step!" << endl;
    }
    virtual void Speak() const {
        cout << "What does a mammal speak? Mammilian!" << endl;
    }

    protected:
    int itsAge;
};
```

Once you have completed class `Mammal` and `Dog`, test them using the following `main` function.

```
int main () {
   Mammal *pDog = new Dog;
   pDog->Move();
   pDog->Speak();

   //Dog *pDog2 = new Dog;
   //pDog2->Move();
   //pDog2->Speak();
}
```

What does it output, is that what you expected? Remove the keyword **`virtual`** from the class mammal and try it again. Now what happens? Next, put in another pointer to `pDog2` in the main program, but this time make it a pointer to a `Dog`, not a mammal and create a new dog. Now what happens? What you should realize is that by making the method `Speak` virtual, we can have a little different behavior through dynamic (runtime) binding.

(b) Develop additional classes for `Cat`, `Horse`, and `Mouse` overriding the move and speak methods. Next, test with the modified main:

```
int main () {
    Mammal* theArray[5];
    Mammal* ptr;

    int choice, i;
    for (i = 0; i<5; i++) {
        cout << "(1)dog (2)cat (3)horse (4)mouse: ";
        cin >> choice;
        switch (choice) {
            case 1: ptr = new Dog; break;
            case 2: ptr = new Cat; break;
            case 3: ptr = new Horse; break;
            case 4: ptr = new Mouse; break;
            default: ptr = new Mammal; break;
        }
        theArray[i] = ptr;
    }
    for (i=0;i<5;i++)
        theArray[i]->Speak();

    // Always free dynamically allocated objects
    for (i=0;i<5;i++)
        delete theArray[i];
}
```

Some things to note:

If the `Dog` object had a method, `WagTail()`, which is not in the `Mammal`, you could not use the pointer to `Mammal` to access that method (unless you cast it to be a pointer to `Dog`). Because `WagTail()` is not a **`virtual`** function, and because it is not in a `Mammal` object, you can't get there without either a `Dog` object or a `Dog` pointer to the `Dog` object!!!

The **`virtual`** function magic (polymorphic behavior) operates only on pointers and references. Passing an object by value will not enable the virtual functions to be invoked.

Exercise 2 ....

Implement a class called `Tool`. It should have an **`int`** field called `strength` and a **`char`** field called `type`. You may make them either **`private`** or **`protected`**. The `Tool` class should also contain the function **`void`** `setStrength(`**`int`**`)`, which sets the `strength` for the `Tool`.

Create 3 more classes called `Rock`, `Paper`, and `Scissors`, which inherit from `Tool`. Each of these classes will need a constructor which will take in an **`int`** that is used to initialize the `strength` field. The constructor should also initialize the `type` field using `'r'` for `Rock`, `'p'` for `Paper`, and `'s'` for `Scissors`.

These classes will also need a **`public`** function **`bool`** `fight(Tool)` that compares their strengths in the following way:

- Rock's `strength` is doubled (temporarily) when fighting scissors, but halved (temporarily) when fighting paper.
- In the same way, paper has the advantage against rock, and scissors against paper.
- The `strength` field shouldn't change in the function, which returns **`true`** if the original class wins in `strength` and **`false`** otherwise.

You may also include any extra auxiliary functions and/or fields in any of these classes. Run the program using the following `main` function, and verify that the results are correct.

```
int main() {
    // Example main function
    // You may add your own testing code if you like

    Scissors s1(5);
    Paper p1(7);
    Rock r1(15);
    cout << s1.fight(p1) << p1.fight(s1) << endl;
    cout << p1.fight(r1) << r1.fight(p1) << endl;
    cout << r1.fight(s1) << s1.fight(r1) << endl;

    return 0;
}
```

Exercise 3 ....

This exercise is a text based task. You do not need to write any program/C++ code: the answer should be written in solution3.txt (and might include code fragments if questions ask for them).

(a) Given is the following class hierarchy:

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

class A {
public:
    std::string name;
    A(std::string _name) : name(_name) {}
    virtual void say_hello() { std::cout << "A says hi to " << name << "\n"; }
    void say_bye() { std::cout << "A says bye to " << name << "\n"; }
};

class B : public A {
public:
    B(std::string _name) : A(_name) {}
    void say_hello() { std::cout << "B greets " << name << "\n"; }
};
```

```
class C : public A {
public:
    C(std::string _name) : A(_name) {}
    void say_bye() { std::cout << "C say goodbye to " << name << "\n"; }
};
```

For each of the following functions, specify for each call whether it is polymorphic, i.e., the dynamic type determines the function to be called, or not.

i.
```
void f() {
    A x("Jane");
    x.say_hello(); // call 1
    B y("John");
    y.say_hello(); // call 2
    x = y;
    x.say_hello(); // call 3
}
```

ii.
```
void g() {
    A* x = new A("Jane");
    (*x).say_hello(); // call 1
    B* y = new B("John");
    (*y).say_hello(); // call 2
    x = y;
    (*x).say_hello(); // call 3
}
```

iii.
```
void h() {
    A* x = new A("Jane");
    (*x).say_bye(); // call 1
    C* y = new C("John");
    (*y).say_bye(); // call 2
    x = y;
    (*x).say_bye(); // call 3
}
```

(b) All the following code fragments use operator **delete** and **delete[]** to deallocate memory, but not appropriately. This can either lead to an error or to a memory leak. Find the mistake in each code fragment, explain whether it results in a memory leak or an error, and in the case of an error, point out the location at which it occurs.

i.
```
class A {
public:
    A(unsigned int sz) {
```

```
            ptr = new int[sz];
        }
        ~A() {
            delete ptr;
        }
        /* copy constructor, assignment operator, public methods. */
        ...
    private:
        int* ptr;
    };
```

ii.

```
    struct llnode {
        int value;
        llnode* next;
    };

    void recursive_delete_linked_list(llnode* n) {
        if (n != nullptr) {
            delete n;
            recursive_delete_linked_list(n->next);
        }
    }
```

iii.

```
    class A {
    public:
        A() {
            c = new Cell;
            c->subcell = new int(0);
        }
        ~A() {
            delete c;
        }
        /* copy constructor, assignment operator, public methods */
        ...
    private:
        struct Cell {
            int* subcell;
        };
        Cell* c;
    };
```

iv.

```
    void do_something(int* p) {
        /* Do something */
```

```
    ...
}
void f() {
    int v;
    int* w = &v;
    do_something(w);
    delete w;
}
```

v.

```
class Vec {
public:
    Vec(unsigned int sz) {
        array = new int[sz];
    }
    ~Vec() {
        delete[] array;
    }
    int& operator[](int l) {
        return array[l];
    }
    /* copy constructor, assignment operator, other public methods */
    ...
private:
    int* array;
};

void f() {
    Vec v(5);
    delete[] &v[0];
}
```