Excercise set week Eight

Feedback on these excercises is very much appreciated. Send mail to one of the lecturers or to Aryan at aryannm@gmail.com.

This one will be especially relevant to implementing inheritance in your projects.

Problem X - Repetition

1. Create a class `Zombie` and subclass it by creating classes `ArmoredZombie` and `FeralZombie`.
2. Create member methods and member variables as you see fit for these classes (be creative!).

Problem One - Static Polymorphism

The Traps - Diamond Of Death

Refer to google or wikipedia.

1. Create a third class, `ArmoredFeralZombie` which inherits from `FeralZombie` and 'ArmoredZombie

```
class ArmoredFeralZombie : public ArmoredZombie, public FeralZombie {
}
```

2. Does this compile? Why not?
3. Make the code compile (hint virtual inheritance)

The Traps - Early Binding

Suppose your class Zombie has a member method

```
void act(){
    std::cout<<"Zombie Attack!"<<std::endl;
}
```

Implement the same function in the base classes `ÀrmoredZombie` and `FeralZombie`but with correct output (eg "Feral Zombie Attack!").

Now create an instance of `ArmoredZombie` in your main function. Test `act()`.

Great! It shows the correct output, now suppose you have a function (not a member of a class, just a standalone function)

```
void provokeAttack(Zombie& z){
    z.act();
}
```

Now create an instance of each type, eg:

```
    Zombie z;
    ArmoredZombie a;
    FeralZombie f;
    ArmoredFeralZombie af;

    provokeAttack(z);
    provokeAttack(a);
    provokeAttack(f);
    provokeAttack(af);
```

1. What is the expected output?
2. What gets printed out acutally?

This is unexpected for Java programmers, it seems the base class function is being called, no matter what, but in C++ this is perfectly valid behavior.

Problem Two - Runtime Polymorphism

The Early Binding Problem

Technically it isn't a problem but what we really meant to do when we created functions with the same name and return type `void act()` in the different classes was to use different behavior.

How do we achieve this? By telling the compiler that it should use the runtime type. As to why this is called runtime polymorphism, see the following example.

```
class A{
};
class B : public A {
};
class C : public A {
};
```

And somewhere (maybe main)

```
A* a;
int coinflip = std::rand();
if(coinflip > 1) a = new B();
if(coinflip <=1) a = new C();
```

The compiler cannot predict the future. It has no way to know that `a` will be either B or C before the coin is actually tossed and if we want different behavior depending on whether it is catually `B` or `C` we have to declare its methods as `virtual` which tells the compiler on use of `a.foo()` (supposing it had a foo function) that it should use the foo in A or the foo in B depending on what type as is pointing to!

1. Now that you know this, change the classes above so that you actually get correct output from `provokeAttack`

Problem Three - Runtime & Static Trap - Slicing

Now that we have different classes for different types of Zombies we would like to have one big container for all the zombies. Knowing that a `FeralZombie` has a `is-a` relationship with `Zombie` (directly inherits it) we can try to create a vector of `Zombies` and store every zombie there!

1. Create this vector 'std::vector
2. Create a Zombie, a FeralZombie, an ArmoredZombie and an ArmoredFeralZombie and add these to the newly created vector.
3. Using a for loop, use `provokeAttack` on every zombie in the vector.
4. Check the output, something is odd..try to explain

What is happening here is called Slicing, for Java developers this behavior is weird but considering that C++ is all about efficiency, zero overhead and memory management (you only pay for what you want) it isn't that weird.

When we create a vector of `Zombie` and put zombies inside, we can't put the original zombie, what happens is that a copy is being made but we have told the compiler to store only zombies here, and so it allocates memory for a zombie, a `FeralZombie` is a Zombie, but it is also MORE than a Zombie and the compiler

cannot know that index 0 will be a normal zombie, index 3 will be a FeralZombie etc which all require different allocation of memmory.

The solution? The compiler cuts off everything off the tail of the baseclass, so every zombie pushed in there will only have the baseclass `àct`.

How do we avoid this? By not storing values (and we can't use references) but rather pointers, all pointers, no matter what they point to, are the same size.

5. Knowing the above, fix your code so it works correctly.

Problem Four - Static Polymorphism & Templates

Suppose you want to store a value inside a class `Box`.

```
class Box {
    private:
        int m_value;
    public:
        Box(int value) : m_value{value}{
        }

        int value() const{
            return m_value;
        }
};
```

1. Create an instance of this box for storing the number `42` and print it out using the `value()` function.
2. Create an instance of this box for storing the number `42.5` and print it out using the value() function.

What happens? You sent in a float or a double and it gets stored as an int, you lose information. Furthermore this box cannot hold a `string` or a `Zombie` so this is not a very good box.

3. Now create another class FloatBox that can hold floats.
4. You see that for every time we need to keep something different in the box we have to create a new class, all we do is type on the keyboard.

What if there was some way for the C++ language to automate this for us? There is! They are called templates and their main function is to automatically generate a new class everytime our class template is being used for something new.

Now consider

```
template<typename T>
class Box{
    T m_value;
    public:
        Box(T value) : m_value{value}{
        }

        T value() const{
            return m_value;
        }
};
```

This is a box that can hold anything!

5. Using this template, create boxes that hold floats, integers, vectors and box of boxes of floats and box of boxes of integers and box of boxes of vector.

This is the power of templates, to be able to autogenerate code at compile that, and the template

Function Templates

You have kind of already seen a function template, the `T value() const` of `Box`.

But consider the following scenario, you are to create functions called add that can take two ints and return an int, that can take two strings and return a string, etc..

We know we can do `int add(int a, int b)` and `float add(float a, float b` etc but this gets cumbersome the more addition operations we have to support.

Now consider the following function

```
template<typename T>
T add(T a, T b){
    return a + b;
}
```

The compiler will gnerate a new function everytime you try to add something new!

1. Try this out with different additions
2. Now create an add function that can take any two types (including both being equal) and add them together.

Remember how I told you this is compile time? That means the more work you can do on your computer when compiling the less you have to do when you ship your `.exe` to your customers, your code will be doing less runtime calculations the more work you can offload to compile-time.

3. (Extra:)Create a function that returns the power of `N^M` using templates.