An Overview of Templates

Templates

- Code using angle brackets e.g. `vector<double>` is written using templates.
- Template programming is quite tricky. It is better to use object orientation where possible.

The problem

```
inline double findMax( double x, double y ) {
    if (x<y) {
        return x;
    } else {
        return y;
    }
}

inline float findMax( float x, float y ) {
    if (x<y) {
        return x;
    } else {
        return y;
    }
}
```

The solution

Templates allow us to write this without breaking the Once and Only Once principle.

```
template <typename T>
inline T findMax(T x, T y) {
    if (x < y) {
        return y;
    }
    else {
        return x;
    }
}
```

Testing it works

```
void testMax() {
    ASSERT( findMax(3, 1)==3); // ints
    ASSERT( findMax(2.0, 3.0) == 3.0); // doubles
    ASSERT( findMax( string("ant"),
            string("zoo")) == string("zoo"));
}
```

Declaration and definition

- Template definitions should be in the same file as the declaration.
- For library functions, this means the definition goes in the header file.

Template classes

```
template <typename T>
class SimpleVector {
public:
    /*  Constructor */
    SimpleVector(int size);
    /*  Destructor */
    ~SimpleVector() {
        delete[] data;
    }
    /*  Access data */
    T get(int index) {
        return data[index];
    }
    /*  Access data */
    void set(int index, T value) {
        data[index] = value;
    }

private:
    T* data;
    /*  Rule of three - we make these private */
    SimpleVector(const SimpleVector& o);
    SimpleVector& operator=(const SimpleVector& o);
};
```

Implementation of a member

```
template <typename T>
SimpleVector<T>::SimpleVector( int size ) {
    data = new T[size];
}
```

- This must go in the header file.

Checking it works

```
void testSimpleVector() {
    SimpleVector<double> v1(3);
    v1.set(1, 2.0);
    ASSERT(v1.get(1) == 2.0);

    SimpleVector<int> v2(3);
    v2.set(1, 2);
    ASSERT(v2.get(1) == 2);

    SimpleVector<string> v3(3);
    v3.set(1, "Test string");
    ASSERT(v3.get(1) == "Test string");

}
```

More template parameters

- You can have multiple template parameters.
- You can use numeric constants as template parameters.

```
void testArray() {
    array<int, 3> a;
    a[2] = 1;
    ASSERT(a[2] == 1);
}
```

YAGNI

- You Aren't Going to Need It (YAGNI).
- Resist the urge to use templates everywhere.
- Do you really need a matrix class of floats?

Templates instead of interfaces?

```
template <typename Option, typename Model>
double monteCarloPrice(
        const Option& option,
        const Model& model,
        int nScenarios = 10000) {
    double total = 0.0;
    for (int i = 0; i<nScenarios; i++) {
        std::vector<double> path = model.
            generateRiskNeutralPricePath(
            option.maturity,
            1);
        double stockPrice = path.back();
        double payoff = option.payoff(stockPrice);
        total += payoff;
    }
    double mean = total / nScenarios;
    double r = model.riskFreeRate;
    double T = option.maturity - model.date;
    return exp(-r*T)*mean;
}
```

- Assumes option has function `payoff`.
- Assumes option has field `maturity`.
- Assumes model has function `generateRiskNeutralPricePath`.

Testing it works

```
void testMonteCarloPricer() {
    CallOption c;
    c.strike = 110;
    c.maturity = 1;
    BlackScholesModel model;
    model.stockPrice = 100;
    model.drift = 0;
    model.riskFreeRate = 0.1;
    model.volatility = 0.2;
    model.date = 0;

    double price = monteCarloPrice(c, model);
    ASSERT_APPROX_EQUAL(price, c.price(model), 0.1);
}
```

Advantages of interfaces

- Interfaces explicitly list requirements of your classes.
- Templates force you to document requirements separately.
- Interfaces tell you that you've got a function wrong when you try to compile your class. They correctly identify where the error is.
- Template approach tells you about errors when you use your class. They incorrectly identify where the error is.
- With templates, all the code is in the header. This makes encapsulation hard
- Templates take longer to build.

Summary

- Templates provide a solution to the Once and Only Once principle that is very useful for writing data structure classes.
- Using templates to write unnecessarily generic code violates the YAGNI principle (You Aren't Going to Need It).
- Templates can be used as an alternative to virtual functions, but virtual functions are normally the better approach.