

# C++ Programming Exercises

## Association

Write a program that represents a squash league in which teams play each other. Define and implement a `League` class that holds references to all the teams and can print out the current league table. Define a `Team` class to hold each team's results (number of games won, drawn and lost).

To make things more interesting, hide the `League` class inside the `Team` class as a static member; then, in the main program you will not have to explicitly refer to the `League` class.

The main program should be on the lines:

```
int main()
{
    Team spurs("Spurs");
    Team rangers("Rangers");
    Team rockets("Rockets");
    Team stingray("Stingray");
    spurs >> rangers;
    rockets >> spurs;
    rockets >> stingray;
    stingray >> spurs;
    Team::printTable();
}
```

where,

1. `printTable()` method calls a corresponding `print()` function in the `League` class.
2. `operator>>()` is overloaded such that the left hand team beats the right hand team.

Note that the above design will mean that the `Team` class will refer to the `League` class and vice-versa. To avoid problems with a single pass compiler you will have to define each class in separate `.cpp` and `.h` files.

## Operator Overloading

Define and implement a `Matrix` class that represents an N by M matrix using the `vector` class from the STL. Overload the operators: `+` `-` `*` to provide basic matrix operations and provide `print()` and `transpose()` methods. A suitable test harness is:

```
int main()
{
    Matrix product;
    Matrix a = { {{1,2,3},{4,5,6}} };
    Matrix b = { {{10,20},{30,40},{50,60}} };
    product = a * b;
    product.print();
    cout << endl;

    Matrix m1{ {{11,12,13},
                {21,22,23},
                {31,32,33},
                {41,42,43}} };
    Matrix m2{ {{2,3,4},
                {5,6,7},
                {9,8,7},
                {10,2,4}} };
    m1.transpose();
    m1.print();
    cout << endl;
    m1.transpose();
    m1.print();
    cout << endl;

    Matrix m = m1 + m2;
    m.print();
}
```

# Inheritance

Create an inheritance hierarchy where `BankAccount` is the base class and `CreditAccount` and `SavingsAccount` are two separate derived classes.

Create the `CreditAccount` class with methods:

```
deposit()
withdraw()
display()
```

with attributes:

```
name
accountNumber
balance
overdraft_limit
```

Create the `SavingsAccount` class with methods:

```
deposit()
display()
apply_interest()
```

with attributes:

```
name
accountNumber
balance
interest_rate
```

Some of the above attributes and methods can be hoisted to the base class - decide which.

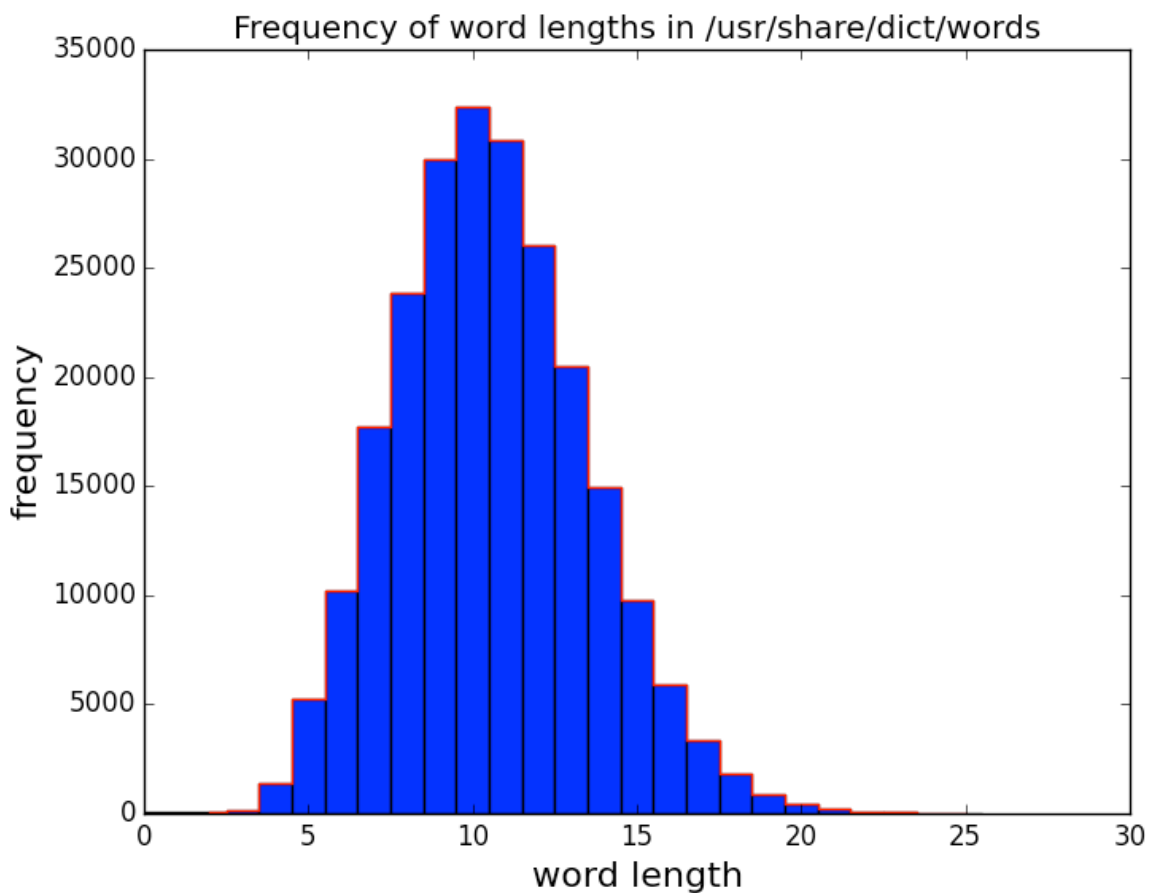
Use a `Money` class to represent the balance in these classes. The `CreditAccount` must prevent account holders from exceeding their overdraft limit. The `SavingsAccount` should apply interest to the current balance each year,. Use the following test harness:

```
int main()
{
    CreditAccount john("John", Money(100, 50), Money(250, 0));
    CreditAccount susan("Susan", Money(50, 10), Money(100, 0));
    SavingsAccount thomas("Thomas", Money(5250, 25), 1.5);
    SavingsAccount christine("Christine", Money(1000, 0), 1.0);
    john.deposit(Money(125, 15));
    john.deposit(Money(125, 40));
    john.display();
    try
    {
        susan.withdraw(Money(2, 99));
        susan.withdraw(Money(152, 25));
        susan.withdraw(Money(152, 25));
    } catch (InsufficientFundsException& e) {
        cout << "Insufficient funds, ";
        susan.display();
    }
}
```

```
thomas.applyInterest();  
thomas.applyInterest();  
thomas.applyInterest();  
thomas.display();  
}
```

## Wordlength Frequencies

Read the file `/usr/share/dict/words` and print a histogram of the occurrence of each size (in percent), giving output similar to the one below:



Obviously we don't have access to Matplotlib, so numerical results will suffice.

## Roman Numerals

Create two functions that convert a decimal number in the range 1 to 4000 to Roman numerals, and back. Use the `map` class from the STL to define two lookup tables to simplify the code:

```
DecimalToRomanMap lookup = {
    {0, ""},
    {1, "I"},
    {2, "II"},
    {3, "III"},
    {4, "IV"},
    {5, "V"},
    {6, "VI"},
    {7, "VII"},
    {8, "VIII"},
    {9, "IX"},
    {10, "X"},
    {20, "XX"},
    {30, "XXX"},
    {40, "XL"},
    {50, "L"},
    {60, "LX"},
    {70, "LXX"},
    {80, "LXXX"},
    {90, "XC"},
    {100, "C"},
    {200, "CC"},
    {300, "CCC"},
    {400, "CD"},
    {500, "D"},
    {600, "DC"},
    {700, "DCC"},
    {800, "DCCC"},
    {900, "CM"},
    {1000, "M"},
    {2000, "MM"},
    {3000, "MMM"},
    {4000, "MMMM"}
};

RomanToDecimalMap symbols = {
    {'M', 1000},
    {'D', 500},
    {'C', 100},
    {'L', 50},
    {'X', 10},
    {'V', 5},
    {'I', 1}
};
```

## Popular Stems

Word prefixes are also called stems. Write a program that reads the file `/usr/share/dict/words` that has one word per input line and finds the most popular stems of size 2 upwards (if you get a tie, just pick one). Your results should be similar to:

```
2: un -> 20358
3: non -> 7804
4: over -> 4041
5: inter -> 1889
6: quasi- -> 1026
7: counter -> 777
8: anthropo -> 133
9: straight- -> 60
10: philosophi -> 31
11: anthropomor -> 26
14: anthropomorp -> 26
13: anthropomorph -> 26
14: anthropomorphi -> 18
15: transcendentali -> 11
16: overintellectual -> 9
17: electroencephalog -> 8
18: electroencephalogr -> 8
19: electroencephalogra -> 8
```