Solutions, C++ Programming Examination

2010–04–13

1. Friendship with templates is tricky. The following declarations are necessary:

```
template <typename T> class Matrix;
template <typename T> ostream& operator<<(ostream& os, const Matrix<T>& m);
template <typename T> vector<T> operator*(const Matrix<T>& lhs, const vector<T>& rhs);
```

The friend trickery continues in the class.

```
template <typename T>
class Matrix {
    friend ostream& operator<< <>(ostream& os, const Matrix<T>& m);
    friend vector<T> operator*<>(const Matrix<T>& lhs, const vector<T>& rhs);
public:
    Matrix(size_t m, size_t n);
    vector<T>& operator[](size_t i) {
        return val[i];
    }
    const vector<T>& operator[](size_t i) const {
        return val[i];
    }
    Matrix& operator+=(const Matrix& rhs);
private:
    vector<vector<T> > val;
};

template <typename T>
Matrix<T>::Matrix(size_t m, size_t n) : val(m) {
    for (size_t i = 0; i != m; ++i) {
        val[i] = vector<T>(n);
    }
}

template <typename T>
Matrix<T>& Matrix<T>::operator+=(const Matrix<T>& rhs) {
    for (size_t i = 0; i != val.size(); ++i) {
        for (size_t j = 0; j != val[i].size(); ++j) {
            val[i][j] += rhs.val[i][j];
        }
    }
    return *this;
}

template <typename T>
Matrix<T> operator+(const Matrix<T>& lhs, const Matrix<T>& rhs)  {
    Matrix<T> ret = lhs;
    return ret += rhs;
}
```

```
template <typename T>
vector<T> operator*(const Matrix<T>& lhs, const vector<T>& rhs) {
    vector<T> res(lhs.val.size());
    for (size_t i = 0; i != lhs.val.size(); ++i) {
        for (size_t j = 0; j != lhs.val[i].size(); ++j) {
            res[i] += lhs.val[i][j] * rhs[j];
        }
    }
    return res;
}

template <typename T>
ostream& operator<<(ostream& os, const Matrix<T>& m) {
    for (size_t i = 0; i != m.val.size(); ++i) {
        for (size_t j = 0; j != m.val[i].size(); ++j) {
            os << m[i][j] << " ";
        }
        os << endl;
    }
    return os;
}
```

2. The iterator class:

```
template <typename T>
class Iterator {
public:
    Iterator(Matrix<T>* ipm, int ii, int jj) : pm(ipm), i(ii), j(jj) {}
    T& operator*() {
        return pm->val[i][j];
    }
    Iterator& operator++() {
        ++j;
        if (j == pm->val[0].size()) {
            ++i;
            j = 0;
        }
        return *this;
    }
    bool operator!=(const Iterator& rhs) const {
        return ! (pm == rhs.pm && i == rhs.i && j == rhs.j);
    }
private:
    Matrix<T>* pm;
    size_t i;
    size_t j;
};
```

Additions to the `Matrix` class (`friend` anywhere, the other three under `public`):

```
    friend class Iterator<T>;
    typedef Iterator<T> iterator;
    iterator begin() {
        return iterator(this, 0, 0);
    }
    iterator end() {
        return iterator(this, val.size(), 0);
    }
```

3. Add a declaration of the function `get_symbol` under `private` in the class.

```
string MorseTable::translate(const string& code) const {
    istringstream is(code);
    string one_code;
    string result;
    while (is >> one_code) {
        result += get_symbol(one_code);
    }
    return result;
}

char MorseTable::get_symbol(const string& one_code) const {
    Node* current = root;
    for (size_t i = 0; i != one_code.length(); ++i) {
        if (one_code[i] == '.') {
             current = current->left;
        } else {
             current = current->right;
        }
        if (current == 0) {
            return '?';
        }
    }
    return current->symbol;
}
```

4.
```
template<typename InputIterator1, typename InputIterator2, typename OutputIterator,
        typename StrictWeakOrdering>
OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2, OutputIterator result,
        StrictWeakOrdering comp) {
    while (first1 != last1 && first2 != last2) {
        if (comp(*first1, *first2)) {
            ++first1;
        } else if (comp(*first2, *first1)) {
            ++first2;
        } else {
            *result = *first1;
            ++first1;
            ++first2;
            ++result;
        }
    }
    return result;
}

template<typename InputIterator1, typename InputIterator2, typename OutputIterator>
OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2, OutputIterator result) {
    typedef typename iterator_traits<InputIterator1>::value_type value_type;
    return set_intersection(first1, last1, first2, last2, result,
            less<value_type>());
}
```