Solutions, C++ Programming Examination

2011–03–08

1.
```
struct compare {
    bool operator()(const string& s1, const string& s2) const {
        return s1.size() < s2.size() || (s1.size() == s2.size() && s1 < s2);
    }
};

int main() {
    using namespace std;
    set<string, compare> words((istream_iterator<string>(cin)),
            istream_iterator<string>());
    copy(words.begin(), words.end(), ostream_iterator<string>(cout, "\n"));
}
```

Note: the extra pair of parentheses around the first argument to the set constructor is necessary ... This is an example of what Scott Meyers called C++'s "most vexing parse" (see `http://en.wikipedia.org/wiki/Most_vexing_parse`).

2.
```
class Skyline {
public:
    Skyline() {}
    void add(size_t left, size_t right, size_t height);
    vector<pair<size_t, size_t> > getAsSequence() const;
private:
    vector<size_t> heights;
};

void Skyline::add(size_t left, size_t right, size_t height) {
    if (right > heights.size()) {
        heights.resize(right);
    }
    for (size_t i = left; i != right; ++i) {
        if (height > heights[i]) {
            heights[i] = height;
        }
    }
}

vector<pair<size_t, size_t> > Skyline::getAsSequence() const {
    vector<pair<size_t, size_t> > sequence;
    size_t prev = 0;
    for (size_t i = 0; i != heights.size(); ++i) {
        if (heights[i] != prev) {
            sequence.push_back(make_pair(i, heights[i]));
            prev = heights[i];
        }
    }
    sequence.push_back(make_pair(heights.size(), 0));
    return sequence;
}
```

3.
```
template <typename T>
class rai {
    friend rai rbegin<T>(T* x, int sz);
    friend rai rend<T>(T* x);
public:
    rai() : ptr(0) {}
    rai& operator++() {
        --ptr;
        return *this;
    }
    T& operator*() {
        return *ptr;
    }
    T* operator->() {
        return ptr;
    }
    bool operator!=(const rai& rhs) {
        return ptr != rhs.ptr;
    }
private:
    T* ptr;
    rai(T* p) : ptr(p) {}
};

template <typename T>
rai<T> rbegin(T* x, int sz) {
    return rai<T>(x + sz - 1);
}

template <typename T>
rai<T> rend(T* x) {
    return rai<T>(x - 1);
}
```

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

Note the return statement: it copies the remainder of the first or the second range to the destination range.

5.
```
class Accumulator {
    friend ostream& operator<<(ostream& s, const Accumulator& a);
public:
    Accumulator() : sum(0) {}
    Accumulator& operator+=(int nbr) {
        history.push(nbr);
        sum += nbr;
        return *this;
    }
    void undo() {
        if (! history.empty()) {
            sum -= history.top();
            history.pop();
        }
    }
    void commit() {
        while (! history.empty()) {
            history.pop();
        }
    }
    void rollback() {
        while (! history.empty()) {
            undo();
        }
    }
private:
    int sum;            // the current sum
    stack<int> history; // numbers added since commit
};

ostream& operator<<(ostream& os, const Accumulator& a) {
    return os << a.sum;
}
```