LUND INSTITUTE OF TECHNOLOGY Department of Computer Science

Solutions, C++ Programming Examination

2014–04–29

1.
```
struct Compare {
    bool operator()(const string& s1, const string& s2) const {
        string::size_type sz = min(s1.size(), s2.size());
        for (string::size_type i = 0; i != sz; ++i) {
            char c1 = tolower(s1[i]);
            char c2 = tolower(s2[i]);
            if (c1 < c2) {
                return true;
            } else if (c2 < c1) {
                return false;
            }
        }
        return s1 < s2;
    }
};

int main() {
    map<string, set<size_t>, Compare> index;
    string separators(",.;:!?()@#");

    size_t line_nbr = 0;
    string line;
    while (getline(cin, line)) {
        ++line_nbr;
        istringstream iss(line);
        string word;
        while (iss >> word) {
            auto it = remove_if(word.begin(), word.end(),
                [&separators](char ch) { return separators.find(ch) != string::npos; });
            word.erase(it, word.end());
            index[word].insert(line_nbr);
        }
    }

    for (const auto& e : index) {
        cout << e.first << " ";
        copy(e.second.begin(), e.second.end(), ostream_iterator<size_t>(cout, " "));
        cout << endl;
    }
}
```

2. a)
```
void f(const Vector v1) { // the copy constructor
    Vector v2(100);                              // the constructor Vector(size_t)
    for (size_t i = 0; i != v2.size(); ++i) {    // size()
        cout << v1[i] << " " << v2[i] << endl; // operator[] const, operator[]
    }
    v2 += v1;                                    // operator+=
    v2 = v1 + v2;                                // operator=, operator+
}
```

b) The class doesn't have a destructor. The memory that is allocated in the constructor will never be released, which leads to a memory leak.

c) `size()` cannot be used for `const` objects. Change to `size_t size() const { return n; }`.

d)
```
Vector& Vector::operator+=(Vector rhs) {
    for (size_t i = 0; i != n; ++i) {
        v[i] += rhs.v[i];
    }
    return *this;
}

Vector operator+(Vector v1, Vector v2) {
    Vector tmp = v1;
    return tmp += v2;
}
```

e) The non-const version of `operator[]` must return a reference so the result can be used on the left-hand side of an assignment: `int& operator[](size_t i) { return v[i]; }`.

f) The constructor `Vector(int)` is used to create a temporary object with 100 elements, which is copied to `v2`. The implicit conversion from `int` to `Vector` can be prevented by specifying the constructor as `explicit`.

g) The memory for the existing vector (on the left-hand side of the assignment) must be deallocated, otherwise there's a memory leak. Also, if `"v1 = v1"` (assignment to self) is performed, the function will allocate memory and then copy from this uninitialized memory.

```
Vector& Vector::operator=(Vector rhs) {
    if (&rhs == this) {
        return *this;
    }
    delete[] v;
    v = new int[n = rhs.n];
    for (size_t i = 0; i != n; ++i) {
        v[i] = rhs.v[i];
    }
    return *this;
}
```

h) The parameters to `operator=`, `operator+=` and `operator+` are called by value, i.e., the arguments are copied to local variables on each call. This is not necessary — it is better to copy only a reference to the arguments. Corrections:

```
class Vector {
    // ...
    Vector& operator=(const Vector&);
    Vector& operator+=(const Vector&);
};

Vector operator+(const Vector&, const Vector&);
```

3.
```
template <typename T>
class OS {
public:
    OS(ostream& o, const char* d = nullptr) : out(&o), delim(d) {}
    OS& operator=(const T& t) {
        *out << t;
        if (delim != nullptr) {
            *out << delim;
        }
        return *this;
    }
    OS& operator*() { return *this; }
    OS& operator++() { return *this; }
    OS& operator++(int) { return *this; }
private:
    ostream* out;
    const char* delim;
};
```

4.
```
#include <iostream>
using namespace std;

struct X {
    bool operator==(int) const { return true; } // either this
    operator int() { return ++value; }          // or this
    int value = 0;                              //
};

int main() {
    X x;
    cout << boolalpha << (x == 1 && x == 2) << endl;
}
```

5. a) When objects are erased from the student vector (`students.erase(it)`) the iterator `it` is invalidated and cannot be used anymore. But `erase` returns an iterator to the object after the erased object, and the statement should be `it = students.erase(it)`.

   b) When an object is erased from the vector the subsequent objects must be moved.

   c)
```
auto it = stable_partition(students.begin(), students.end(), passed_exam);
vector<Student> failed(it, students.end());
students.erase(it, students.end());
```