LUND INSTITUTE OF TECHNOLOGY Department of Computer Science

Solutions, EDAF30 Programmering i C++

2015–08–18

1.
```
bool isVowel(char ch) {
    return ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'y' ||
               ch == 'A' || ch == 'E' || ch == 'I' || ch == 'O' || ch == 'Y';
}

string hyphenate(const string& s) {
    if (s.length() <= 2)
        return s;
    string res(1, s[0]);
    for (size_t i = 1; i < s.length() - 1; ++i) {
        if (! isVowel(s[i]) && isVowel(s[i + 1]))
            res += '-';
        res += s[i];
    }
    res += s[s.length() - 1];
    return res;
}
```

2. Use a map to get logarithmic behaviour (efficient), use the STL algorithm 'transform' to convert the words to lower case:

```
int main() {
    typedef map<string, size_t>::iterator miter;
    map<string, size_t> m;
    string nm;
    cin >> nm;
    ifstream infile(nm);
    string s;
    while (nm >> s) {
        transform(s.begin(), s.end(), s.begin(), ::tolower);
        miter it = m.find(s);
        if (it != m.end())
            it->second++;
        else
            m.insert(make_pair(s, 1));
    }
    for (miter it = m.begin(); it != m.end(); ++it)
        cout << it->first << " " << it->second << endl;
}
```

3.
```
class Polynomial {
    friend ostream& operator<<(ostream& os, const Polynomial& p);
public:
    Polynomial();
    void add_term(double c, size_t d);
    double operator()(double x) const;
    Polynomial& operator+=(const Polynomial& rhs);
```

```
private:
    typedef pair<double, size_t> term_type;
    list<term_type> terms;
};

Polynomial::Polynomial() {}

void Polynomial::add_term(double c, size_t d) {
    if (c == 0) {
        return;
    }
    list<term_type>::iterator it = terms.begin();
    while (it != terms.end() && it->second < d) {
        ++it;
    }
    terms.insert(it, term_type(c, d));
}

double Polynomial::operator()(double x) const {
    double val = 0;
    list<term_type>::const_iterator it = terms.begin();
    for (; it != terms.end(); ++it) {
        val += it->first * pow(x, static_cast<int>(it->second));
    }
    return val;
}

Polynomial& Polynomial::operator+=(const Polynomial& rhs) {
    list<term_type>::iterator lit = terms.begin();           // lhs iterator
    list<term_type>::const_iterator rit = rhs.terms.begin(); // rhs iterator
    while (lit != terms.end() && rit != rhs.terms.end()) {
        if (lit->second < rit->second) {
            ++lit;
        } else if (lit->second > rit->second) {
            terms.insert(lit, *rit);
            ++rit;
        } else {
            double new_coeff = lit->first + rit->first;
            if (new_coeff != 0) {
                lit->first = new_coeff;
                ++lit;
            } else {
                lit = terms.erase(lit);
            }
            ++rit;
        }
    }
    while (rit != rhs.terms.end()) {
        terms.push_back(*rit);
        ++rit;
    }
    return *this;
}

ostream& operator<<(ostream& os, const Polynomial& p) {
    list<Polynomial::term_type>::const_iterator it = p.terms.begin();
    for (; it != p.terms.end(); ++it) {
        os << it->first << "\t" << it->second << endl;
    }
    return os;
}
}
```

4. a)

```
template<typename T>
T *bin_sok(T& sokt, T *forsta, T *sista)
{
    if (forsta>sista)
        return 0;
    T *mitten = forsta + (sista-forsta)/2;
    if (sokt<*mitten)
        return bin_sok(sokt,forsta,mitten-1);
    else if (sokt>*mitten)
        return bin_sok(sokt,mitten+1,sista);
    else
        return mitten;
};
```

b) Lägg till följande kod i klassen `MI6Agent`:

```
public:
     bool operator< (MI6Agent& a) {
        return *agent_number<*a.agent_number;
    }
    bool operator> (MI6Agent& a) {
        return *agent_number>*a.agent_number;
    }
```

c) En minnesläcka innebär att vi inte frigör minne som inte längre behövs. Det kan då inte återanvändas till andra saker senare och om programmet kör tillräckligt länge kommer minnet förr eller senare att ta slut. I vårt fall är det minnet för strängobjekten som innehåller agentnummer och namn som inte frigörs när ett MI6Agent-objekt frigörs.

d) För att undvika minnesläckor och inte samtidigt riskera att frigöra minne för tidigt behöver vi lägga till en destruktor, en kopieringskonstruktor och en tilldelningsoperator som som ser till att strängobjekten frigörs respektive kopieras vid behov. Lägg därför till följande kod i klassen `MI6Agent`:

```
public:
   ~MI6Agent() {
        delete agent_number;
        delete agent_name;
    }
    MI6Agent(const MI6Agent& b) {
        agent_number = new string(*b.agent_number);
        agent_name = new string(*b.agent_name);
    }
    MI6Agent& operator=(MI6Agent& b) {
        if (this!=&b) {
            delete agent_number;
            delete agent_name;
            agent_number = new string(*b.agent_number);
            agent_name = new string(*b.agent_name);
        }
        return *this;
    }
```

e) Det hade varit enklare att ändra klassen `MI6Agent` så att attributen `agent_number` och `agent_name` är av typen `string` i stället för `string *`. Man får då också ändra på motsvarande sätt i set- respektive get-metoderna (samt i jämföreleoperatorerna).