LUND INSTITUTE OF TECHNOLOGY Department of Computer Science

# Solutions, EDAF30 Programmering i C++

## 2015–01–14

1.
```
string encrypt(const string& text, unsigned int key) {
    srand(key);
    string result(text.size(), ' ');
    for (string::size_type i = 0; i != text.size(); ++i) {
        int r = static_cast<int>(256.0 * rand() / (RAND_MAX + 1));
        result[i] = (text[i] + r) % 256;
    }
    return result;
}
```

2. a) The parameter to `print` is called by value, which means that the actual parameter `v1` is copied to the function by the standard copy constructor. The pointer `v1.v` is copied, and there are two pointers to the same array. When the function is exited the destructor for the copy is executed, and the memory to which `v1.v` points is marked as free.

   When `v2` is allocated the deallocated memory is used again. To correct the error you must write your own copy constructor:

```
Vector(const Vector& rhs) : v(new int[rhs.sz]), sz(rhs.sz) {
    for (size_t i = 0; i != sz; ++i) {
        v[i] = rhs.v[i];
    }
}
```

   b) Additions to the class:

```
T& operator[](size_t i) { return v[i]; }
const T& operator[](size_t i) const { return v[i]; }
```

   A global function:

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

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);
    Polynomial operator+(const Polynomial& lhs, 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;
}

Polynomial operator+(const Polynomial& lhs, const Polynomial& rhs) {
    Polynomial res = lhs;
    return res += rhs;
}
```

4. We to choose to use a set to store the words we encounter. An alternative would be to use a linked list or similar data structure, but we would then have to check for each new word if it is already present in the list or not.

```
int main(int argc, const char * argv[]) {
    string fname;
    cout << "File name:";
    cin >> fname;
    set<string> words;

    ifstream f(fname);
    string w;
    while (f >> w) {
        words.insert(w);
    }
    cout << "Number of words: " << words.size() <<endl;
    return 0;
}
```