LUND INSTITUTE OF TECHNOLOGY Department of Computer Science

Solutions, EDAF30 Programmering i C++

2015–05–06

1.
```
class Urand {
public:
        Urand(size_t n, unsigned int seed);
        size_t rnd();
private:
        vector<int> nbrs; // the numbers
        size_t used; // number of used numbers
};

Urand::Urand(size_t n, unsigned int seed) : nbrs(n) {
        srand(seed);
        for (size_t i = 0; i != nbrs.size(); ++i) {
                nbrs[i] = i;
        }
        random_shuffle(nbrs.begin(), nbrs.end());
        used = 0;
}

size_t Urand::rnd()  {
        if (used == nbrs.size()) {
                random_shuffle(nbrs.begin(), nbrs.end());
                used = 0;
        }
        return nbrs[used++];
}
```

2. Här visas en implementation som bygger på adapterklassen `stack`, men det går lika bra att använda en länkad lista eller en vektor. I det senare fallet får man även hålla reda på hur många tal man stoppat in i vektorn och om vektorn tar slut får man allokera en större vektor och kopiera över innehållet.

```
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;
}
```

3.
```
std::string::size_type Editor::findMatchingLeftPar(std::string::size_type pos) const {
    char rightPar = text[pos];
    char leftPar;
    switch (rightPar) {
        case ')' : leftPar = '('; break;
        case ']' : leftPar = '['; break;
        default : leftPar = '{'; break;
    }
    int level = 0;
    bool found = false;
    string pars = string(1, leftPar) + string(1, rightPar);
    std::string::size_type i = text.find_last_of(pars, pos - 1);
    while (i != std::string::npos && ! found) {
        char ch = text[i];
        if (ch == leftPar && level == 0) {
            found = true;
        } else {
            if (ch == rightPar) {
                level++;
            } else if (ch == leftPar) {
                level--;
            }
            i = text.find_last_of(pars, i - 1);
        }
    }
    return i;
}
```

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 `Bil`:

```
public:
     bool operator< (Bil& b) {
        return *reg_nr<*b.reg_nr;
    }
    bool operator> (Bil& b) {
        return *reg_nr>*b.reg_nr;
    }
```

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 registreringsnummer och ägarnamn som inte frigörs när ett Bil-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 `Bil`:

```
public:
   ~Bil() {
        delete reg_nr;
        delete owner;
    }
    Bil(const Bil& b) {
        reg_nr = new string(*b.reg_nr);
        owner = new string(*b.owner);
    }
    Bil& operator=(Bil& b) {
        if (this!=&b) {
            delete reg_nr;
            delete owner;
            reg_nr = new string(*b.reg_nr);
            owner = new string(*b.owner);
        }
        return *this;
    }
```

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