Lösningsförslag, Tentamen i C++

2016–08–16

1. a)

```
template<typename T>
T *bin_sok(const T& sokt, T *forsta, T *sista)
{
    if (forsta>sista)
        return nullptr;
    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< (const Bil& b) const {
        return *reg_nr<*b.reg_nr;
    }
    bool operator> (const Bil& b) const {
        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=(const Bil& b) {
        if (this!=&b) {
            delete reg_nr;
            delete owner;
            reg_nr = new string(*b.reg_nr);
            owner = new string(*b.owner);
        }
        return *this;
    }
```

Fr o m C++-11 bör man även lägga till en *move-konstruktor* och en *move-tilldelningsoperator*, analogt.

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).

2. a) Parametern till `print` värdeanropas, och eftersom inte klassen `Vector` definierar en kopieringskonstruktor används den automatiskt genererade, som bara kopierar värdena på medlemsvariabler rakt av. Detta betyder att *pekaren* `v1.v` kopieras, och att det nu finns två pekare som pekar på samma array. När funktionen returnerar körs destruktorn på kopian och minnet som `v1.v` pekar på avallokeras.

Nu är `v1.v` en *dangling pointer* ("kvardröjande pekare"), och minnet den pekar på kan skrivas över vid nästa minnesallokering. Lösningen är att definiera en egen kopieringskonstruktor:

```
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) Tillägg till klassen:

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

En fri funktionsmall (`operator<<` kan inte vara medlemsfunktion i klassen `Vector` eftersom första parametern är en `ostream&`.) :

```
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;
}
```

Notera att funktionen inte behöver vara `friend` eftersom den bara använder det publika gränssnittet. Att ha funktionsmallar som `friend` till en klassmall är lite krångligt syntaktiskt.

3. Det som händer är ett så kallat *buffer overflow*. Strängen `"abcdefgh"` är (inklusive terminerande null) 9 tecken lång, men arrayen den skrivs till har bara plats till åtta tecken, och här råkar null (0) skriva över värdet på `nbr`.

4. 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.

```
#include <stack>
#include <iostream>

using std::ostream;
using std::stack;

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;
}
```

5. a) Klassen `Pet` är *abstrakt* eftersom den har en *rent virtuell* funktion, `speak()`, och man kan inte skapa objekt av en abstrakt klass.

Eftersom en `std::vector<Pet>` skulle innehålla element av en abstrakt klass, vilka man inte kan skapa, är detta inte tillåtet.

b) För polymorfa klasser behöver man använda pekare (eller referenser, men sådana kan inte lagras i en `std::vector`).

```
void test_pets()
{
    Dog d;
    Cat c;
    Bird b;

    std::vector<Pet*> l;

    l.push_back(&b);
    l.push_back(&c);
    l.push_back(&d);

    for(const Pet* p : l) {
        cout << p->speak() << endl;
    }
}
```