Dr. M. Grabert Abteilung Angewandte Informationsverarbeitung 13. Dezember 2001
Johannes Mayer Blatt 6

Objektorientierte Programmierung mit C++ (WS 2001/2002)

Abgabetermin: 13. Dezember 2001

Beispiellösung

7 Sichere Vektoren (3 Punkte)

Bei der Klasse `vector<T>` fällt ein Zugriff außerhalb der Array-Grenzen gar nicht auf! Dies muss sich ändern und mit Vererbung ist das ja auch kein Problem – hoffen wir mal. Also schreiben Sie eine Klasse `svector<T>`, die von `vector<T>` erbt und den Operator `[]` neu implementiert, so dass bei Zugriff außerhalb der Array-Grenzen eine `out_of_range`-Exception geworfen wird. Ansonsten soll diese Klasse dieselben Methoden wie `vector<T>` besitzen.

Lösung:

svector.cc

```
#include "svector.h"

main()
{
    svector<int> arr(5);

    try {
        arr[1] = 3;
        cout <<arr[1]<<endl;
        cout <<arr[5]<<endl;
    }
    catch (out_of_range ex) {
        cerr << ex.what() << endl;
    }
}
```

svector.h

```
// Wrapper Class fuer den Vektor-Template
// Operator[] prueft nun auf gueltigen Arrayzugriff
```

```
#ifndef _svector_h
#define _svector_h
#include <vector>
#include <stdexcept>

template <class T>
class svector : public vector<T> {
public:
        svector<T>(int size = 0):vector<T>(size) { }
        T& operator[](int pos) {
                if (pos >= size() || pos < 0)
                        throw out_of_range("pos out of range");
                return vector<T>::operator[](pos);
        }
};
#endif
```

8 Wer führt das Protokoll? (4 Punkte)

Viele Programme machen irgendwelche Ausgaben in Log-Files. Dies können wir natürlich schön kapseln durch die Klasse `log`, die alles in eine Datei schreibt. Soweit der *Stand*. Jetzt kommt Ihnen noch die Idee, dass die Logs auch (zusätzlich) auf die Standardausgabe geschrieben werden sollen. Also bilden Sie eine Unterklasse von `log`, die dies übernimmt. Natürlich könnte man auch zählen, wieviele Logs ausgegeben wurden und diese über die Methode `size()` verfügbar machen. Also schreiben Sie von der abgeleiteten Klasse wiederum eine Unterklasse, die `size()` implementiert. Aber nicht genug. Ihr Chef hat gehört, dass man Vererbung mittels „Delegation“ simulieren kann, d. h. Sie erzeugen eine neue Klasse (ohne Vererbung) und geben Ihr ein Objekt ihrer „Oberklasse“. Aufrufe von „ererbten“ Methoden reicht diese Klasse einfach an das „Oberklassen“-Objekt weiter. Schreiben Sie also – damit sich Ihr Chef freut – noch die beiden Klassen mittels Delegation (und ohne Vererbung).
Beispiel zu Delegation:

```
class A {
public:
    A() {}
    void method();
}
class B : class A {
public:
    B() : A() {}
}
```

Dann besitzt die Klasse `B` die von `A` ererbte Methode `method()`. Man kann `B` aber auch mittels Delegation realisieren.

```
class B_delegate {
    A a;
```

```
public:
    B_delegate() : a() {}
    void method() { a.method(); }
}
```

Dabei muss man natürlich „ererbte“ Methoden definieren. Jedoch bestehen Sie nur aus einem Aufruf der entsprechenden Methode des `A`-Objektes `a`.

Lösung:

log.cc

```
#include <iostream>
#include <fstream>
#include <string>

class log {
    ofstream out;
    string logfile;
public:
    log(char *file) : out(file), logfile(file) {}
    void append(string s)
    {
        if (out)
            out << "> " << s << endl;
    }
    string filename() { return logfile; }
};

class mylog_inherit : public log {
public:
    mylog_inherit(char *file) : log(file) {}
    void append(string s) { cout << "> " << s << endl; log::append(s); }
};

class mylog_delegate {
    log l;
public:
    mylog_delegate(char *file) : l(file) {}
    void append(string s) { cout << "> " << s << endl; l.append(s); }
    string filename() { return l.filename(); }
};

class mycountlog_inherit : public mylog_inherit {
    int count;
public:
    mycountlog_inherit(char *file) : mylog_inherit(file), count(0) {}
    void append(string s) { count++; mylog_inherit::append(s); }
    int size() { return count; }
};
```

```
class mycountlog_delegate {
    mylog_delegate l;
    int count;
public:
    mycountlog_delegate(char *file) : l(file), count(0) {}
    void append(string s) { count++; l.append(s); }
    string filename() { return l.filename(); }
    int size() { return count; }
};

void main() {
    mycountlog_delegate l("mylog.txt");

    l.append("test1");
    l.append("test2");
    cout << "Laenge:  " << l.size() << endl;
    cout << "Logfile: " << l.filename() << endl;
}
```

9 Wir brauchen einen Verwalter! (3 Punkte)

Sie bekommen eine (minimale) Klasse *student* ähnlich wie die Klasse `student` aus der Vorlesung. Jeder Student wird eindeutig über sein Login identifiziert. Außerdem stehen bei jedem Studenten seine C++-Punkte. Alles Unwichtige haben wir weggelassen :-)). Nun sollen Sie die Studenten, die erzeugt und wieder zerstört werden, verwalten. Wie im `main`-Teil des Beispielprogrammes zu sehen ist, sollen Sie zumindest drei Methoden implementieren, mit denen der Punktedurchschnitt aller Studenten berechnet werden kann, alle Studenten (d. h. alle momentan existierenden Studenten-Objekte) ausgegeben werden können und ein Studenten-Objekt zu einem gegebenen Login gesucht werden kann. Wie man in `main` sieht, werden die angelegten Studenten nicht explizit in den Verwalter aufgenommen oder aus ihm wieder entfernt. Dies soll im Konstruktor bzw. Destruktor passieren.

Lösung:

student.cc

```
#include <string>
#include <map>
#include <iostream>

class student {
    string login;
    string name;
    string vorname;
    int punkte;

    static map<string,student*> studis;
```

```
    static void add(student &s) {
        studis[s.login] = &s;
    }
    static void remove(student &s) {
        if (studis[s.login] == &s)
            studis.erase(s.login);
    }
public:
    student(string login, string name, string vorname, int punkte) :
        login(login), name(name), vorname(vorname), punkte(punkte)
    {
        add(*this);
    }
    ~student()
    {
        remove(*this);
    }
    string getlogin() { return login; }
    string getname() { return name; }
    string getvorname() { return vorname; }
    int getpunkte() { return punkte; }

    static double mittlerePunkte()
    {
        double sum = 0.0;
        int count = 0;
        for (map<string,student*>::iterator i = studis.begin(); i != studis.end(); i++)
            count++, sum += i->second->punkte;
        return sum / count;
    }
    static void printAll()
    {
        for (map<string,student*>::iterator i = studis.begin(); i != studis.end(); i++)
            cout << *(i->second) << endl;
    }
    static student *find(string login)
    {
        if (studis.find(login) != studis.end())
            return studis[login];
        else
            return 0;
    }

    friend ostream &operator<<(ostream&, const student&);
};

// statische Klassenelemente muessen hier noch definiert (!) werden
map<string,student*> student::studis;
```

```
ostream &operator<<(ostream &out, const student &s) {
    return out << s.login << ":" << s.name << ":" << s.vorname << ":" << s.punkte;
}

void main() {
    student s1("joe", "Mayer", "Josef", 12);
    student s2("tom", "Berger", "Thomas", 15);
    student s3("franz", "Luidl", "Franz", 14);
    student s4("fw", "Wehr", "Friedrich", 11);
    student s5("kalle", "Wurm", "Karl-Heinz", 17);

    cout << "Mittelwert: " << student::mittlerePunkte() << endl;

    cout << "---------------------------------" << endl;

    student::printAll();

    cout << "---------------------------------" << endl;

    string login;
    cout << "Login: ";
    cin >> login;

    student *s = student::find(login);
    if (s != NULL)
        cout << *s << endl;
    else
        cout << "Es gibt keinen Studenten mit dem Login " << login << "." << endl;
}
```