Objektorientierte Systeme 1 - SWB2 & TIB2
Hausaufgabe 5

Aufgabe 1: Virtuelle Methoden

Schauen Sie sich das nachfolgende Programm an und geben Sie an, was das Programm ausgibt.

```
#include <iostream>
using namespace std;

class A {
public:
    virtual void f() {
        cout << "A::f() - ";
        g();
    }
    virtual void g() {
        cout << "A::g()" << endl;
    }
    virtual void h(int) {
        cout << "A::h(int)" << endl;
    }
};

class B : public A {
public:
    virtual void h(char) {
        cout << "B::h(char)" << endl;
    }
};

class C : public B {
public:
    virtual void f() {
        cout << "C::f() - ";
        A::f();
    }
    virtual void g() {
        cout << "C::g()" << endl;
    }
    virtual void h(int) {
        cout << "C::h(int)" << endl;
    }
    virtual void h(char) {
        cout << "C::h(char)" << endl;
    }
};
```

```
int main() {
    C c;
    A * aptr = &c;
    B & bref = c;

    cout << " 1. "; c.f();
    cout << " 2. "; c.A::f();
    cout << " 3. "; aptr->f();
    cout << " 4. "; (*aptr).f();
    cout << " 5. "; ((A)(*aptr)).f();
    cout << " 6. "; bref.f();
    cout << " 7. "; dynamic_cast<B*>(aptr)->f();
    cout << " 8. "; B(bref).f();
    cout << " 9. "; bref.h(1);
    cout << "10. "; c.h('a');
    cout << "11. "; aptr->h('a');
    cout << "12. "; bref.h('a');

    return 0;
}
```

Aufgabe 2: Polymorphie und Modularisierung

Von der Klasse Angestellte können keine Objekte instantiiert werden. Sie stellt aber vier Methoden bereit, von denen `double einkommen()` eine rein abstrakte Methode ist.

Von der Klasse `Angestellte` werden zwei Klassen abgeleitet: `Boss` und `Arbeiter`. Diese Klassen redefinieren die von der Klasse `Angestellte` geerbten Methoden `einkommen()` und `print()`.

Das Programm ist auf sieben Dateien verteilt: `Angestellte.hpp`, `Boss.hpp`, `Arbeiter.hpp`, `Angestellte.cpp`, `Boss.cpp`, `Arbeiter.cpp` und `main.cpp`.

Ergänzen Sie die Programmteile an den Stellen, die mit /* HIER */ gekennzeichnet sind, so dass das Programm funktioniert.

```
// Datei: Angestellte.hpp

#pragma once

class Angestellte
{
    char * vorname;
    char * nachname;
public:
    Angestellte(const char *, const char *);
    ~Angestellte();
```

```
    const char *getVorname() const;
    const char *getNachname() const;
    /* HIER */  einkommen() /* HIER */ ;
    virtual void print() const;
};
```

```
// Datei: Boss.hpp

#pragma once

#include "Angestellte.hpp"

class Boss : public Angestellte
{
    double bossGehalt;
public:
    Boss(const char *, const char *, double = 0.0);
    void setBossGehalt( double );
    virtual double einkommen() const;
    virtual void print() const;
};
```

```
// Datei: Arbeiter.hpp

#pragma once

#include "Angestellte.hpp"

class Arbeiter : public Angestellte
{
    double stundenLohn;
    double stunden;
public:
    Arbeiter(const char *, const char *, double = 0.0, double = 0.0);
    void setStundenLohn( double );
    void setStunden(double);
    virtual double einkommen() const;
    virtual void print() const;
};
```

```
// Datei: Angestellte.cpp

#include <iostream>
#include <string>
using namespace std;
/* HIER */

Angestellte::Angestellte(const char * vn, const char * nn)
```

```
{
    vorname = new char[strlen(vn) +1];
    strcpy(vorname, vn);
    nachname = new char[strlen(nn) +1];
    strcpy(nachname, nn);
}

Angestellte::~Angestellte()
{
    /* HIER */
    /* HIER */
}

const char *Angestellte::getVorname() const
{
    return vorname;
}

const char *Angestellte::getNachname() const
{
    return nachname;
}

void Angestellte::print() const
{
    cout << vorname << ", " << nachname << endl;
}
```

```
// Datei: Boss.cpp

#include <iostream>
using namespace std;
/* HIER */

Boss::Boss( const char * vn, const char * nn, double g)
    : Angestellte(vn, nn)
{
    setBossGehalt( g );
}

void Boss::setBossGehalt( double g)
{
    bossGehalt = (g > 0) ? g : 0;
}

/* HIER */  Boss::einkommen() /* HIER */
{
    return bossGehalt;
```

```
}

void Boss::print() const
{
    cout << "\nBoss; ";
    Angestellte::print();
}
```

```
// Datei: Arbeiter.cpp

#include <iostream>
using namespace std;
/* HIER */


Arbeiter::Arbeiter(const char *vn, const char * nn,
                   double stdl, double std)
: Angestellte(vn, nn)
{
    setStundenLohn(stdl);
    setStunden(std);
}

void Arbeiter::setStundenLohn( double stdl)
{
    stundenLohn = (stdl>0) ? stdl : 0;
}

void Arbeiter::setStunden(double std)
{
    stunden = (std>=0)? std : 0;
}

/* HIER */  Arbeiter::einkommen() /* HIER */
{
    if (stunden <= 40) return stundenLohn * stunden;
    else return 40*stundenLohn + (stunden-40)*stundenLohn*1.5;
}

void Arbeiter::print() const
{
    cout << "\nArbeiter: ";
    Angestellte::print();
}
```

```
// Datei: main.cpp

#include <iostream>
```

```
using namespace std;
/* HIER */


void virtualMitPointern(const Angestellte *);
void virtualMitReferenz(const Angestellte &);

int main()
{
    Boss b("Gerd", "Mayer", 1000.00);
    b.print();
    cout << "Gehalt: " << b.einkommen() << endl;
    virtualMitPointern(&b);
    virtualMitReferenz(b);

    Arbeiter a("Rainer", "Mueller", 10.00, 40);
    a.print();
    cout << "Lohn: " << a.einkommen() << endl;
    virtualMitPointern(&a);
    virtualMitReferenz(a);
    return 0;
}

void virtualMitPointern(const Angestellte * angstP)
{
    angstP->print();
    cout << "Gehalt: " << angstP->einkommen() << endl;
}

void virtualMitReferenz(const Angestellte & angstR)
{
    angstR.print();
    cout << "Gehalt: " << angstR.einkommen() << endl;
}
```

Aufgabe 3: Eine eigene Vector-Klasse

In der Klasse `MyString` aus Hausaufgabe 2 haben Sie ein dynamisches Array genutzt, um den String so zu speichern und zu verarbeiten, dass der Nutzer nicht mit den Schwierigkeiten von C-Strings und dynamischen Arrays kämpfen muss. Es wäre doch hilfreich, nicht nur für den Typ **`char`** sondern für alle möglichen Datentypen eine vergleichbar einfache Behandlung von dynamischen Arrays zu haben und diese nur einmalig zu implementieren. Dies machen wir hier mit einer eigenen Klasse `MyVector`.

Legen Sie Kopien Ihrer Dateien `MyString.hpp` und `MyString.cpp` an und nennen Sie sie in `MyVector.*` um. Führen Sie dann die folgenden Änderungen und Ergänzungen durch.

a) Legen Sie eine Datei `MyVectorData.hpp` an, in der die Klasse `MyVectorData` deklariert wird. Diese Klasse dient dazu die Vaterklasse für alle Klassen zu sein, deren Objekte wir im

Vektor speichern können.

b) Die Klasse `MyVectorData` hat einen virtuellen Destruktor, der gar nichts macht.

c) Darüberhinaus hat die Klasse die zwei rein virtuellen Methoden:

```
virtual MyVectorData * clone() const = 0;
virtual void print(bool=true) const = 0;
```

Die Methode `clone()` dient später dazu, Objekte des korrekten Typs zu konstruieren.

d) Zusätzlich hat die Klasse einen virtuellen Zuweisungsoperator:

```
virtual MyVectorData & operator=(const MyVectorData &) {
    return *this;
}
```

e) Nennen Sie die Klasse `MyString` konsequent überall in `MyVector` um.

f) In der Klasse `MyVector` wollen wir nun auf ein dynamisches Array verweisen. Dies ist nun nicht mehr ein **`char`**-Array, da wir ja Objekte der Klasse `MyVectorData` speichern wollen. Um jedoch die Polymorphie nutzen zu können, können wir die Objekte nicht direkt speichern sondern speichern nur Pointer auf Objekte abgeleiteter Klasse. Ersetzen Sie daher **`char`** `* strPtr` durch `MyVectorData ** strPtr`. Wir haben hier also ein dynamisches Array von Pointern auf Objekte der Klasse `MyVectorData` (vgl. Abb. 1).

g) Löschen Sie den Konvertierkonstruktor sowie die Methoden `assign(`**`const char`** `*)` und `c_str()`.

h) Ändern Sie den Rückgabetyp der Methode `at(`**`unsigned int`**`)` sinnvoll ab.

i) Schreiben Sie eine Methode **`void`** `push_back(`**`const`** `MyVectorData& d)`, die eine Kopie von `d` am Ende des Vectors (d.h. an Position `size`) speichert. Nutzen Sie dazu die Methode `MyVectorData * clone()` **`const`**, die eine Kopie von `d` auf dem Heap anlegt.

j) Schreiben Sie eine Methode `print()`, die auf jedem Datenelement des Vektors eine `print`-Methode aufruft.

k) Nehmen Sie dann alle noch notwendigen Änderungen vor. Beachten Sie z.B. beim Destruktor sowie bei der Methode `clear()`, dass Sie auch die durch die Methode `clone()` erzeugten Objekte der Klasse `MyVectorData` löschen müssen. Denken Sie daran, dass Sie die C-Stringfunktionen `strncpy` etc. nicht mehr nutzen können, da Sie nicht Zeichen sondern Pointer kopieren müssen.

Testen Sie Ihre Klasse `MyVector` mit dem nachfolgenden Hauptprogramm, indem Sie Ihre Klasse `DrawingObject` von der Klasse `MyVectorData` ableiten und in der Klasse `Point`

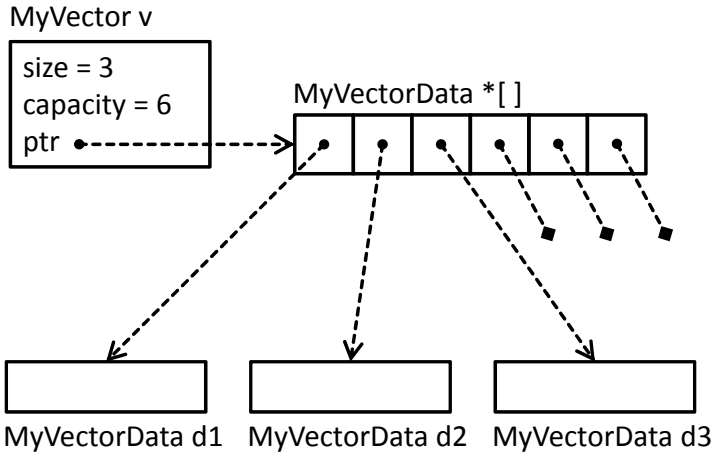

Abbildung 1: Das Speicherlayout eines Objektes der Klasse `MyVector`

die notwendigen Änderungen vornehmen. Dazu gehört auch die Implementierung des virtuellen Zuweisungsoperators als Redefinition des Zuweisungsoperators der Klasse `MyVectorData`. Für die Klasse `Point` sieht dieser z.B. so aus:

```
Point & Point::operator=(const MyVectorData & p) {
    const Point * ptr =
       dynamic_cast<Point*>(const_cast<MyVectorData*>(&p));
    x = ptr->x;
    y = ptr->y;
    return *this;
}
```

```
#include <iostream>
#include "MyVector.hpp"
#include "Circle.hpp"
using namespace std;

int main() {
    MyVector v1;
    Point p1(1,1);
    Point p2(2,2);
    Point p3(3,3);
    Point p4(4,4);
    v1.push_back(p1);
    v1.push_back(p2);
    v1.push_back(p3);
    v1.at(1).print();
    v1.at(2).print();
    v1.at(1) = v1.at(2);
    v1.print();
    MyVector v2(v1);
    v2.print();
    dynamic_cast<Point&>(v1.at(2)).move(10,10);
    v1.print();
    v2.print();
    cout << "Groesse von v1: " << v1.size();
    cout << "  Kapazitaet von v1: " << v1.capacity() << endl;
    if (!v1.empty()) {
        v1.clear();
    }
    cout << "Groesse von v1: " << v1.size();
    cout << "  Kapazitaet von v1: " << v1.capacity() << endl;
    v1.print();
    v2.print();
    cout << "Groesse von v2: " << v2.size();
    cout << "  Kapazitaet von v2: " << v2.capacity() << endl;
    v2.push_back(p1);
    cout << "Groesse von v2: " << v2.size();
    cout << "  Kapazitaet von v2: " << v2.capacity() << endl;
```

```
    v2.print();
    v1.push_back(p1);
    v2.append(v1);
    v1.print();
    v2.print();
    cout << "Kreise: " << endl;
    MyVector v3;
    Circle c1(p1, 1);
    Circle c2(p2, 2);
    v3.push_back(c1);
    v3.push_back(c2);
    v3.print();
    return 0;
}
```

Hinweis: So nützlich unsere Klasse `MyVector` auch ist, es gibt doch Einschränkungen. Daher gibt es in C++ bereits eine vorprogrammierte viel mächtigere Klasse `vector` aus der Standard Template Library (STL) (siehe http://www.cplusplus.com/reference/vector/vector/).