Fachhochschule Wedel
Prof. Dr. Uwe Schmidt

18. August 1997

Aufgaben zur Klausur **C** und **Objektorientierte Programmierung** im SS 98 (WI h103)

Zeit: 120 Minuten erlaubte Hilfsmittel: keine

Bitte tragen Sie Ihre Antworten und fertigen Lösungen ausschließlich an den gekennzeichneten Stellen in das Aufgabenblatt ein. Ist ihre Lösung wesentlich umfangreicher, so überprüfen Sie bitte nochmals Ihren Lösungsweg.

Sollten Unklarheiten oder Mehrdeutigkeiten bei der Aufgabenstellung auftreten, so notieren Sie bitte, wie Sie die Aufgabe interpretiert haben.

Zwischen der Bearbeitung der Aufgabe 8 über Java und der Aufgabe 9 über C++ kann gewählt werden. Wird die Aufgabe 8 bearbeitet, so wird nur diese gewertet, sonst Aufgabe 9.

Viel Erfolg !

Diese Klausur besteht einschließlich dieses Deckblattes aus 18 Seiten

---

Aufgabe 1:

Bei dem folgenden Programm wird angenommen, daß **long int** in 32 bit und **char** in 8 bit in der Maschine dargestellt werden.

```
#include <stdio.h>

int main(int argc, char * argv[]) {

    union {
        long int l;
        char c[4];
    } cv;

    cv.l = 0x01020304l;

    printf("%d,%d,%d,%d\n",
            cv.c[0],
            cv.c[1],
            cv.c[2],
            cv.c[3]);

    return 0;
}
```

Welches sind die möglichen Ausgaben für dieses Programm?

. . . . . .

. . . . . .

. . . . . .

Aufgabe 2:

Gegeben sei das folgende Programmstück.

```
char *p1, *p2;

void f(void) {

    *++p1 = *++p2;

    *p1-- = *p2--;
}
```

Schreiben Sie den Anweisungsteil der Funktion so um, daß keine Increment– und Decrement–Operatoren verwendet werden und nur eine Zuweisung pro Anweisung ausgeführt wird.

.....

.....

.....

.....

.....

.....

Aufgabe 3:

Gegeben seien die folgenden Variablen:

```
int i;
unsigned int u;
long int l;
double f;
char *cp;
int *ip;
int a[20];
```

Bestimmen Sie für die folgenden Ausdrücke den Typ. Vorsicht: Es kommen fehlerhafte Ausdrücke von. Kennzeichnen Sie diese entsprechend

i + f .....

++f .....

ip[i] .....

a[a[i]] .....

cp + 0x42 .....

cp ? i : f .....

f == 0 .....

! ip .....

~ip .....

cp && cp .....

f += i .....

~l .....

!l .....

l || i .....

l | i .....

Aufgabe 4:

Entwickeln Sie Makros zur Erzeugung von Bitmasken. Die Makros sollen Ausdrücke vom Typ **int** erzeugen. Sie sollen unabhängig von der Zahlendarstellung, 1–er oder 2–er-Komplement, sein.

1. Ein Makro `low_zeroes(n)` zum Setzen der `n` niederwertigen Bits auf 0, alle anderen Bits sollen auf 1 gesetzt werden.

   .....

   .....

   .....

2. Ein Makro
   low_ones(n) zum Setzen der `n` niederwertigen Bits auf 1, alle anderen Bits sollen auf 0 gesetzt werden.

   .....

   .....

   .....

3. Ein Makro
   mid_zeroes(width,offset), das die niederwertigen `offset` Bits auf 1 setzt, die folgenden `width` Bits auf 0, und alle übrigen wieder auf 1.

   .....

   .....

   .....

4. Ein Makro
   mid_ones(width,offset), das die niederwertigen `offset` Bits auf 0 setzt, die folgenden `width` Bits auf 1, und alle übrigen wieder auf 0.

   .....

   .....

   .....

Aufgabe 5:

Gegeben sei das folgende Program

```
#include <stdio.h>

int main(int argc, char *argv[]) {

    int i, sum=0;

    for(i=0; i<10; ++i) {
        switch (i) {
        case 0:
        case 1:
        case 3:
        case 5:
            sum += i+1;
        default:
            continue;
        case 4:
            break;
        }
        break;
    }

    printf("%d\n",sum);

    return 0;
}
```

Welche Ausgabe erzeugt dieses Programm?

.....

Aufgabe 6:

Wann ist es sinnvoll, eine Datenkomponente in einer Klasse mit dem Zugriffsrecht `protected` zu deklarieren?

1. wenn die darauf operierenden Funktionen auch `protected` sind. ja ☐ nein ☐ weiß nicht ☐
2. wenn der Zugriff auf Datenkomponenten aus Effizienzgründen aus abgeleiteten Klassen notwendig ist. ja ☐ nein ☐ weiß nicht ☐
3. wenn Basisklassen Zugriffsrechte haben sollen, der Benutzer aber nicht. ja ☐ nein ☐ weiß nicht ☐
4. wenn abgeleitete Klassen Zugriffsrechte haben sollen, der Benutzer aber nicht. ja ☐ nein ☐ weiß nicht ☐
5. wenn abgeleitete Klassen Zugriffsrechte haben sollen, aber nicht die Basisklassen. ja ☐ nein ☐ weiß nicht ☐
6. wenn der Benutzer Zugriff haben soll, aber abgeleitete Klassen nicht. ja ☐ nein ☐ weiß nicht ☐
7. wenn der Wartungsaufwand für die Basisklassen gering gehalten werden soll. ja ☐ nein ☐ weiß nicht ☐
8. wenn der Wartungsaufwand für abgeleitete Klassen gering gehalten werden soll. ja ☐ nein ☐ weiß nicht ☐
9. wenn nur die direkt aus der Klasse abgeleitete Klasse Zugriffsrechte haben soll, weitere abgeleitete Klassen aber nicht. ja ☐ nein ☐ weiß nicht ☐
10. immer. ja ☐ nein ☐ weiß nicht ☐
11. nie. ja ☐ nein ☐ weiß nicht ☐

Aufgabe 7:

In der Objektorientierten Programmierung spielt die Wiederverwendung eine zentrale Rolle. Dabei können schon entwickelte, fertige Klassen von anderen Klassen beerbt oder benutzt werden.

Vererbung (hier mit Java Syntax)

```
class Y extends X { ... };
```

Benutzung

```
class Y {
    X d;
    ...
};
```

Welches sind Vorteile der Benutzung gegenüber der Vererbung der Klasse X in der Klasse Y:

1.	Die modulare Verständlichkeit nimmt zu.	ja ☐	nein ☐	weiß nicht ☐
2.	Die Speicherverwaltung wird einfacher.	ja ☐	nein ☐	weiß nicht ☐
3.	Der Quellcode wird kürzer.	ja ☐	nein ☐	weiß nicht ☐
4.	Der Objektcode wird kürzer.	ja ☐	nein ☐	weiß nicht ☐
5.	Die Laufzeit des Programms verringert sich.	ja ☐	nein ☐	weiß nicht ☐
6.	Die Compilierzeit verringert sich.	ja ☐	nein ☐	weiß nicht ☐
7.	Die Konstruktoren brauchen nicht neu entwickelt werden.	ja ☐	nein ☐	weiß nicht ☐
8.	Der Quelltext wird selbstdokumentierend.	ja ☐	nein ☐	weiß nicht ☐
9.	Die Erweiterbarkeit ist verbessert.	ja ☐	nein ☐	weiß nicht ☐
10.	Die Flexibilität wird erhöht.	ja ☐	nein ☐	weiß nicht ☐
11.	Die Veränderung der Klasse X hat keinen Einfluß auf Programmcode außerhalb von Y.	ja ☐	nein ☐	weiß nicht ☐
12.	Die Namensgebung für Methoden in Y kann besser an die Aufgabenstellung angepaßt werden.	ja ☐	nein ☐	weiß nicht ☐

Aufgabe 8:

Gegeben sei eine Klassenhierarchie zur Verarbeitung von einstelligen reellwertigen Funktionen, d.h. reelwertige Funktionen werden durch Java–Objekte dargetellt.

Die Schnittstelle für die Funktionen wird als interface realisiert. Sie enthält eine Methode *at* zur Berechnung der Funktion an einer Stelle x. Außerdem ist eine Methode *derive* (= ableiten) zur Berechnug der 1. Ableitung zu implementieren. Drei häufig verwendete Funktionen werden beim Laden der Schnittstelle erzeugt. Diese sind global zugreifbar.

Die Schnittstelle:

```
public
interface Function {

  public
  double at(double x);

  public
  Function derive();

  public static Function sin = new Sine();
  public static Function cos = new Cosine();
  public static Function zero = new ConstFunction(0.0);

}
```

Die Klasse für die Sinus–Funktion (*Sine*), die Klasse für die Kosinus–Funktion (*Cosine*) sei analog definiert.

```
public
final
class Sine implements Function {

  public
  double at(double x) {
    return
      Math.sin(x);
  }

  public
  Function derive() {
    return
      Function.cos;
  }
}
```

Die Klasse für konstante Funktionen (*ConstFunction*):

```
public
final
class ConstFunction implements Function {
  private
  double c;

  public
  ConstFunction(double c) {
    this.c = c;
  }

  public
  double at(double x) {
    return
      c;
  }

  public
  Function derive() {
    return
      Function.zero;
  }
}
```

Erweitern Sie diese Klassenhierarchie um eine Klasse *AddFunction* für Funktionen der Form $f(x) = f_1(x) + f_2(x)$. Hinweis: $f'(x) = f_1'(x) + f_2'(x)$

Erweitern Sie diese Klassenhierarchie um eine Klasse $MultFunction$ für Funktionen der Form $f(x) = f_1(x) * f_2(x)$. Hinweis: $f'(x) = f_1'(x) * f_2(x) + f_1(x) * f_2'(x)$

Entwerfen Sie eine Schnittstelle *FindZero* für Algorithmen, die zu einer Funktion f in einem Intervall x_1 bis x_2 eine Nullstelle suchen.

Ist es sinnvoll, die konkreten Klassen in diesem Beispiel als **final** zu deklarieren?

ja ☐ nein ☐

Begründung:

.....

.....

.....

Kann das **final** Attribut vom Compiler im Allgemeinen zur Verbesserung des JVM–Codes genutzt werden?

ja ☐ nein ☐

Begründung:

.....

.....

.....

Die Schnittstelle *Function* ist als Java–interface deklariert. Ist diese Deklaration von Vorteil gegenüber einer Deklaration als abstrakte Klasse?

ja ☐ nein ☐

Begründung:

.....

.....

.....

Aufgabe 9:

Gegeben sei die folgende (unvollständige) Stringklasse.

```
#include <stdlib.h>
#include <string.h>

class STR {
        char * p;
        void create ( char * p1 );
                // erzeugt eine Kopie von p1
                // auf der Halde und speichert
                // den Zeiger in p
                // oder setzt p auf den 0-Zeiger

        void delet ();
                // gibt den p zugeordneten
                // Speicherbereich auf der Halde frei

        int inv() const { return !p || strlen(p) > 0; }
                // Invariante: die leere Zeichenreihe
                // wird eindeutig durch 0-Zeiger dargestellt
public :
        STR();
                // der default Konstruktor für Initialisierung mit leerer Zeichenreihe

        STR(const char * s);
                // der Konstr. für C strings

        STR(const STR & s);
                // der copy–Konstruktor

        STR & operator = (const STR & s);
                // der Zuweisungsoperator

       ~STR();
                // der Destruktor

        int len () const;
                // string Länge

};
```

Entwickeln Sie die Definitionen der member–Funktionen so, daß Objekte dieser Klasse sich aus Sicht eines Anwenderprogramms wie einfache Daten verhalten und die Speicherverwaltung für dynamische Strukturen vollständig in der Klasse versteckt ist.

Die Hilfsfunktion *create* (Fehlersituationen sollen zu einem Programmabbruch führen):

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Die Hilfsfunktion *delet*:

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Der default–Konstruktor:

. . . . . .

Der char * Konstruktor:

. . . . . .

. . . . . .

. . . . . .

Der copy–Konstruktor:

. . . . . .

. . . . . .

. . . . . .

Der Zuweisungsoperator:

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Der Destruktor:

. . . . . .

. . . . . .

Die *len*–Funktion:

. . . . . .

. . . . . .

. . . . . .

. . . . . .
