Technische Universität Hamburg-Harburg
Institut für Zuverlässiges Rechnen
Prof. Dr. S.M. Rump

SS 2009

Lösungen der Klausur zur Vordiplom-Prüfung Programmiermethodik

29. Juli 2009

Sie haben 90 Minuten Zeit zum Bearbeiten der Klausur.

Tragen Sie bitte zunächst Ihren Namen, Ihren Vornamen, Ihre Matrikelnummer, Ihr Studienfach und Ihr Studiensemester in **DRUCKSCHRIFT** in die folgenden jeweils dafür vorgesehenen Felder ein.

Name:															
Vorname:															
Matr.-Nr.:								Fach				Sem.			

Bei jeder der fünf Aufgaben können 12 Punkte erreicht werden, insgesamt also 60 Punkte. Es sind alle Aufgaben zu bearbeiten. Zum Bestehen der Klausur sind 30 Punkte erforderlich.

Aufg.	Punkte	Korr.
1		
2		
3		
4		
5		
$\sum$		

Bonus	
Note	

Zur Beachtung:
Schreiben Sie bitte weder mit Bleistift noch mit Rotstift. Es sind keinerlei Hilfsmittel in dieser Klausur zugelassen! Das Schreiben vor dem Startsignal und auch das Schreiben nach dem Endsignal führt ohne weitere Warnung sofort zur Ungültigkeit der Klausur. Dies gilt auch für das Schreiben von Namen und Matrikelnummer nach dem Endsignal. Vergessen Sie nicht, den „Vorbehalt“ zu unterschreiben.
Hinweis zum Programmieren:
Programmieren Sie die Aufgaben in ANSI-C

Vorbehalt
Ich bin darüber belehrt worden, dass die von mir zu erbringende Prüfungsleistung nur dann bewertet wird, wenn die Nachprüfung durch das Zentrale Prüfungsamt der TUHH meine offizielle Zulassung vor Beginn der Prüfung ergibt.

_____

(Datum, Unterschrift)

Aufgabe 1 (12 Punkte)

Ein Student sollte ein C-Programm schreiben, mit dem sich die Häufigkeit eines vom Benutzer eingegebenen Wortes in der übergebenen Textdatei zählen lässt. Der Student hat folgendes Programm geschrieben. Nur leider lässt es sich nicht kompilieren. Sein Tutor fand genau 12 syntaktische Fehler im Programm. Finden Sie diese Fehler.

Hinweis: `fgetc` liefert `EOF` als Rückgabewert, wenn das Dateiende erreicht worden ist.

```
1   #define N 128
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include <string.h>
5
6   int main(int argc, char *argv[]) {
7           char *inword, *sword;
8           int i = 0, sum = 0;
9           FILE *wordfile;
10
11          if ( argc < 2 ) {
12                  printf("Falscher Programmaufruf: searchword(filename) erwartet.\n");
13                  return -1;
14          }
15          inword = malloc(sizeof(char)*N);
16          sword = malloc(sizeof(char)*N);
17          wordfile = fopen(argv[1],"r");
18          if (wordfile == NULL) {
19                  printf("Fehler: konnte die Datei %s nicht oeffnen.\n",argv[1]);
20                  return -2;
21          }
22          printf("Bitte geben Sie das gesuchte Wort ein: >>");
23          scanf("%s",sword);
24          while ( i < (N-1) && (inword[i] = fgetc(wordfile)) != EOF ) {
25                  if ( inword[i] == ' ' || inword[i] == '\n' ) {
26                          inword[i] = '\0';
27                          i = 0;
28                          if ( strcmp(inword,sword) == 0 ) {
29                                  sum++;
30                          }
31                  }
32                  else i++;
33          }
34          if ( i >= N ) {
35                  printf("Fehler:\n");
36                  printf("die Datei enthaelt ein Wort laenger als %d Zeichen.\n",N-1);
37                  return -3;
38          }
39          else {
40                  printf("Das Wort %s kommt %d Mal in der Datei vor.\n",sword,sum);
41          }
42
43          return 0;
44  }
45
46
```

(12 Punkte)

Aufgabe 2 (12 Punkte)

Die „Ackermannfunktion“, benannt nach ihrem Erfinder Wilhelm Ackerman[1], dient in der theoretischen Informatik als ein Beispiel für eine Klasse von Funktionen, die berechenbar, aber nicht primitiv-rekursiv sind. Eine wichtige Eigenschaft dieser Funktion ist, dass sie extrem schnell wächst. In vereinfachter Version wird die Ackermannfunktion wie folgt definiert:

$$A(m,n) = \begin{cases} n+1 & \text{falls } m = 0 \\ A(m-1,1) & \text{falls } m > 0 \text{ und } n = 0 \\ A(m-1, A(m,n-1)) & \text{sonst.} \end{cases}$$

a) Schreiben Sie eine **rekursive** Funktion `A(m,n)`, die für gegebene Integer-Werte m und n den Funktionswert der Ackermannfunktion berechnet. Beachten Sie bei der Wahl des Datentyps für den Rückgabeparameter, dass die Funktion sehr schnell große Werte annehmen kann.

a)

```
1   unsigned long ackerman(unsigned short m, unsigned long n) {  /* 2 Punkte */
2           if (m == 0) {                                        /* 1 Punkt */
3                   return n + 1;                                /* 1 Punkt */
4           }
5           if (m > 0 && n == 0) {                               /* 1 Punkt */
6                   return ackerman(m-1,1);                      /* 1 Punkt */
7           }
8           return ackerman(m-1,ackerman(m,n-1));                /* 1 Punkt */
9   }
10
```

(7 Punkte)

b)

1. Möglichkeit: `unsigned long (int)` – größter Wertebereich für positive Integer-Zahlen; `ULONG_MAX = 2^32-1`

2. Möglichkeit: `double` – größere positive Zahlen als mit `unsigned long` erreichbar; durch die begrenzte Stellenanzahl geht die Genauigkeit der Ergebnisse bei sehr großen Zahlen verloren. Es ist möglich, dass das Endergebnis überhaupt nicht mit dem richtigen Ergebnis übereinstimmen wird. (2 Punkte)

c) Nachteile:

Rekursive Lösung ist für große n und m sehr langsam. Das ist das Resultat immer wiederkehrender Prozesse, die in Verbindung mit der Kontextvorbereitung zur Funtkionsausführung stehen. Lokale Variablen müssen auf dem Stack angelegt werden, Initialisierung muss durchgeführt werden usw.. Außerdem wird der Stack stark belastet. Bei hoher Rekursiontiefe, kann es zu Stacküberlauf kommen. (2 Punkte)

Vorteile:

Meistens einfachere Berechnungsvorschrift. Klarere Lösungsstruktur. (1 Punkt)

[1] Wilhelm Ackermann (*29.3.1896 - †24.12.1962) war ein deutscher Mathematiker. Er studierte von 1914 - 1924 mit Unterbrechungen durch den Ersten Weltkrieg Mathematik, Physik und Philosophie an der Universität Göttingen. Er war ein Schüler von David Hilbert in Göttingen und wurde berühmt durch die nach ihm benannte Ackermannfunktion.

Aufgabe 3 (12 Punkte)

a) Betrachten Sie folgendes Programm:

```
1   #include <stdio.h>
2
3   void swp1(int a, int b) {
4           int tmp = a;
5           a = b;
6           b = tmp;
7   }
8
9   void swp2(int *a, int *b) {
10          int tmp = *a;
11          *a = *b;
12          *b = tmp;
13  }
14
15  int main() {
16          int a = 0, b = 1;
17          printf("a = %d, b = %d\n",a,b);
18          swp2(&a,&b); printf("swp2(a,b) ergibt: a = %d, b = %d\n",a,b);
19          swp1(a,b);   printf("swp1(a,b) ergibt: a = %d, b = %d\n",a,b);
20          return 0;
21  }
22
```

Ergänzen Sie die Ausgabe des Programms an angegebenen Stellen.

Ausgabe:

```
17  a = 0, b = 1
18  swp2(a,b) ergibt: a = 1, b = 0
19  swp1(a,b) ergibt: a = 1, b = 0
```

(1 Punkt)

b) Erklären Sie den Unterschied zwischen Call-by-Value und Call-by-Reference.

Call-by-Value: Die übergebenen Parameterwerte sind nur Kopien der originalen Werte. Jegliche Änderungen dieser Werte haben nur lokale Charackter und überhaupt keine Auswirkung auf die Originale.

Call-by-Reference: Die übergebenen Parameter sind Adressen der Originale. Jeglicher Schreib-Zugriff auf die referenzierten Objekte verändert ihren Zustand/Wert. (2 Punkte)

c) Definieren Sie einen *neuen* strukturierten Datentyp `znumber` mit zwei Datenfeldern `real` und `imag` vom Typ `double` zur Speicherung von komplexen Zahlen:

```
1   typedef struct complexnumber{            /* 2 Punkte */
2           double real;                     /* 1 Punkt */
3           double imag;                     /* 1 Punkt */
4   } znumber;
```

(4 Punkte)

Vervollständigen Sie die nachfolgende Funktion zur Summation zweier Komplexen Zahlen vom Typ `znumber`.

```
0   znumber *sum(znumber *z1, znumber *z2) {
1           znumber *newz = malloc(sizeof(znumber)); /* 2 Punkte */
2           newz->real = z1->real + z2->real;        /* 1 Punkt */
3           newz->imag = z1->imag + z2->imag;        /* 1 Punkt */
4           return newz;                             /* 1 Punkt */
5   }
```

(5 Punkte)

Lösung Aufgabe 4 (12 Punkte)

a) Stellen Sie in den ersten drei Zeilen die Dezimalzahlen -432, 234 und -256 im 2-er Komplement (`short int`) binär dar. Die dritte Zeile enthält bereits eine solche Binär-Darstellung. Geben Sie links den entsprechenden Dezimalwert an. Addieren Sie abschließend in der letzten Zeile **alle vier** Zahlen binär. (5 Punkte)

-777 :	1	1	1	1	1	1	0	0	1	1	1	1	0	1	1	1
$+345$:	0	0	0	0	0	0	0	1	0	1	0	1	1	0	0	1
-102 :	1	1	1	1	1	1	1	1	1	0	0	1	1	0	1	0
-84 :	1	1	1	1	1	1	1	1	1	0	1	0	1	1	0	0

1 Punkt

1 Punkt

1 Punkt

1 Punkt

$\sum = -777 + 345 - 102 - 84 = -618$:

1	1	1	1	1	1	0	1	1	0	0	1	0	1	1	0

1 Punkt

b) Welche Ausgabe erzeugt die folgende Code-Sequenz? Nehmen Sie dabei an, dass Fließkommazahlen stets aufgerundet werden („round to infinity“). (4 Punkte)

```
100         short int x=2, y=0, z=4; float f1, f2; double d1, d2;
101         while(x>0){x=x*x; y+=1;}
102         printf("%d \n",x);              /* 1. Ausgabe: 0   (1 Punkt)*/
103         printf("%d \n",y);              /* 2. Ausgabe: 4   (1 Punkt)*/
104         x=100; y=11; x=x/y/z;
105         printf("%d \n",x);              /* 3. Ausgabe: 2   (1 Punkt)*/
106         f1=d1=0.1; d2=f2=0.1;
107         printf("%u \n",d2>d1);          /* 4. Ausgabe: 1   (1 Punkt)*/
```

Begründen Sie Ihre Antwort:

1. Ausgabe: $x = 2^{2^y}$, wrap-around bei $y = 4$, $x = 2^{16} = (2^{15} + 1) + 2^{15} - 1 = -32767 + 32767 = 0$
2. Ausgabe: s.o.
3. Ausgabe: Auswertung von links nach rechts: $x/y/z = (100/11)/4 = 9/4 = 2$
4. Ausgabe: $0.1_d = 0.00011001100...{}_b$, $f2 = 0.1$ wird als `single` bei Aufrundung nach oben größer als die `double`-Variable $d2 = 0.1$, folglich $d2 > d1$.

c) Stellen Sie die Dezimalzahl -34.125 im IEEE-754 Standard dar: (3 Punkte)

Es gilt $-34.125 = -(32 + 2 + 0.125) = -(2^5 + 2^1 + 2^{-3}) = -(1 + 2^{-4} + 2^{-8})2^5 = 1.m \cdot 2^e$,

Vorzeichenbit:

1

1 Punkt

Exponent: (mit dem Shift B=127): $e + B = 5 + 127 = 132 = 128 + 4 = 2^7 + 2^2$

1	0	0	0	0	1	0	0

1 Punkt

Mantisse: Denken Sie an die implizite Eins ! $m = 2^{-4} + 2^{-8}$

0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 Punkt

Lösung Aufgabe 5 (12 Punkte)

a) Ein Unternehmen will die Personalnummern seiner Mitarbeiter in einer verketteten Liste verwalten. Definieren Sie hierfür eine Struktur `sEmpList`, welche die Komponenten `PersNr` für die ganzzahlige Personalnummer und `next` für den Zeiger auf den Nachfolger besitzt. (3 Punkte)

```
100  struct sEmpList{
101    int PersNr;
102    struct sEmpList *next;
103  }; /*sNameList*/
```

b) Schreiben Sie eine Funktion `NewEmployee`, die eine neue Personalnummer am Anfang einer Liste mit Struktur aus a) einfügt. Eingabeparameter sind ein Zeiger `E` auf den Listenanfang und die neue Personalnummer `PersNr`. Rückgabewert soll ein Zeiger auf den Anfang der erweiterten Liste sein. (3 Punkte)

```
104  struct sEmpList *NewEmployee(struct sEmpList *E, int PersNr){
105    struct sEmpList* F;
106    F = (struct sEmpList*) malloc (sizeof(struct sEmpList));
107    F->PersNr=PersNr;
108    F->next=E;  /*neues Element an den Anfang setzen*/
109    return(F);
110  }
```

c) Eine Methode um Zugriffszeiten auf häufig angefragte Personalnummern zu verringern besteht bei sogenannten adaptiven Listen darin, bei jedem Zugriff auf ein Element dieses mit seinem Vorgänger zu vertauschen. Schreiben Sie eine Funktion `swap`, die ein Listenelement mit seinem Vorgänger vertauscht. Sie besitze die gleichen Eingabe- und Rückgabeparameter wie die Funktion `NewEmployee` aus b). (6 Punkte)

```
111  struct sEmpList* swap(struct sEmpList* E,int PersNr){
112    struct sEmpList *F; struct sEmpList *G;
113    if(E!=NULL && E->next!=NULL){ /*Für leere oder 1-elementige Listen ist nichts zu tun.*/
114      F=E;
115      if(E->next->PersNr == PersNr){
116      /* Das gesuchte Listenelement steht an zweiter Stelle, wird daher bei Vertauschung
117         erstes Listenelement und damit zum Listenzeiger.*/
118        E=E->next;
119        F->next =E->next;
120        E->next = F;
121      }
122      else{
123        while(F->next->next != NULL && F->next->next->PersNr != PersNr){F=F->next;}
124        if(F->next->next != NULL){ /*Dann gilt F->next->next->PersNr == PersNr*/
125          G=F->next;
126          F->next=F->next->next;
127          G->next=F->next->next;
128          F->next->next =G;
129        }
130      }
131    }
132    return(E);
133  }
```