Klausur: Programmiersprache C

$10.02.2011, 10^{\underline{00}} - 12^{\underline{00}}$

Name:	
Vorname:	
MatrNr:	Kürzel:

Aufgabe	erreichbare Punktzahl	erreichte Punktzahl
1 Präprozessor	5	
2 Vereinbarungen u. Anweisungen	20	
3 Mehrere Quelldateien	12	
4 Zeiger-Operationen	12	
5 Text kodieren/dekodieren	18	
6 String in Substrings zerlegen	18	
7 Spezielle Funktion `errormsg`	15	
Gesamtpunktzahl	100	

Note:

1 Präprozessor

Geben Sie (durch Ankreuzen) an, welche der folgenden Aussagen über den C-Präprozessor zutreffen.

ja	nein	
☐	☒	Header-Dateien müssen immer am Anfang einer Quelldatei eingefügt werden.
☐	☒	Präprozessor-Direktiven werden zur Laufzeit ausgeführt.
☒	☐	Eine Header-Datei darf weitere Header-Dateien einfügen.
☐	☒	Präprozessor-Direktiven dürfen zusammen mit anderen Anweisungen in einer Zeile kodiert werden.
☒	☐	Jede Präprozessor-Direktive beginnt mit dem Zeichen **#**, dem ausschließlich Zwischenraumzeichen vorausgehen dürfen.

2 Vereinbarungen und Anweisungen

(a) Erzeugen Sie für jede der nachfolgenden Konstanten eine Variable, die exakt den gleichen Typ wie die Konstante besitzt und initialieren Sie die Variable mit der Konstanten:

i. `0.0` `double d = 0.0;`

ii. `0xf` `int i = 0xf;`

iii. `0f` ist keine zulässige Konstante

iv. `"abc"` `char *s = "abc";`

(b) Vereinbaren Sie einen Zeiger, der auf die (in Teilaufgabe (a) erzeugte) Variable mit dem Wert `0.0` verweist, und diese als veränderliches Objekt beschreibt:

```
double *dp = &d;
```

(c) Vereinbaren Sie einen Zeiger, der auf die (in Teilaufgabe (a) erzeugte) Variable mit dem Wert `0.0` verweist, und diese als **un**veränderliches Objekt beschreibt:

```
const double *cdp = &d;
```

(d) Vereinbaren Sie eine Struktur, die 4 Komponenten mit unterschiedlichen Datentypen beinhaltet und initialieren Sie alle Komponenten mit geeigneten Werten:

```
struct s {
  double d;
  int    i;
  float  f;
  char * s;
} obj = { 0.0, 0xf, 0.f, "abc" };
```

(e) Legen Sie dynamisch einen Speicherbereich an, der die gleiche Größe wie die unter (d) erzeugte Struktur hat, und übertragen Sie den Inhalt der Struktur in diesen Bereich:

```
#include <stdlib.h>

struct s *ptr2struct =
  (struct s *) malloc(sizeof(struct s));
*ptr2struct = obj;
```

(f) Erklären Sie jeden der folgenden Code-Schnipsel mit Ihren eigenen Worten.

i. `int i; float f=1.5, *fp = &f; i = (int) *fp;`

Zunächst werden drei Variablen erzeugt:

- `i` ist vom Typ **`int`**.
- `f` ist eine **`float`**-Variable und enthält den Gleitkommawert `1.5`.
- `fp` ist ein Zeiger auf eine **`float`**-Variable und enthält als Wert die Adresse der Variablen `f`.

Durch Dereferenzieren des Zeigers `fp` wird auf den Wert `1.5` der Variablen `f` zugegriffen. Dieser Wert wird durch den Cast-Operator in den **`int`**-Wert `1` umgewandelt. Dieser wird der Variablen `i` zugewiesen.

ii. `int i; float f=1.5, *fp = &f; i = *(int *) fp;`

Die Variablen `i`, `f` und `fp` werden wie oben erzeugt.
Danach wird der Wert der Zeigervariablen `fp` durch den Cast in den Typ „Zeiger auf **`int`**“ umgewandelt, so dass beim anschließenden Dereferenzieren auf die Variable `f` wie auf eine **`int`**-Variable zugegriffen wird. Der so aus `f` ausgelesene Wert wird der Variablen `i` zugewiesen. Die Variablen `i` und `f` enthalten nun identische Bitfolgen.

3 Mehrere Quelldateien

Durch die folgenden drei Dateien ist eine unvollständige Anwendung gegeben. Vervollständigen Sie diese, indem Sie die Teilaufgaben (a) - (c) bearbeiten.

```
/* Datei: solver.h */

#ifndef SOLVER_H_INCLUDED
#define SOLVER_H_INCLUDED

_____

#define DIM 100 _____

_____

extern int errcode; /* Deklaration */ _____

_____

#ifdef INLINE _____

#define getSourceFilename() FILENAME _____

#endif _____

_____

_____

int solver(double [][DIM], int);

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

#endif
```

```
/* Datei: solver.c */

#include <stdio.h>
#include "solver.h"

#define FILENAME "solver.c" _____

_____

#ifndef INLINE _____

static const char *getSourceFilename(void) { _____

  return FILENAME; _____

} _____

#endif _____

_____

int errcode = 0; /* Definition */ _____

int solver(double m[][DIM], int n) {

  printf("%s: Function solver called.\n",
         getSourceFilename());
  if (n > DIM)
    errcode = 3;
  /* . . . */
  return 0;
}

_____

_____

_____

_____

_____

_____

_____

_____

_____
```

```
/* Datei: main.c */

#include <stdio.h>
#include "solver.h"

#define FILENAME "main.c" _____

_____

#ifndef INLINE _____

static const char *getSourceFilename(void) { _____

  return FILENAME; _____

} _____

#endif _____

_____

int main(void) {
  double matrix[DIM][DIM], result;

  /* . . . */
  errcode = 0;
  result = solver(matrix, DIM);
  if (errcode != 0)
    printf("%s: Fehler: errcode = %d\n",
           getSourceFilename(), errcode);
  /* . . . */
  return 0;
}

_____

_____

_____

_____

_____

_____

_____

_____
```

(a) Das in allen drei Dateien verwendete Makro `DIM` ist nicht definiert. Fügen Sie eine Definition des Makros an geeigneter Stelle hinzu, so dass `DIM` jeweils durch den Wert 100 ersetzt wird.

(b) Ergänzen Sie alle notwendigen Vereinbarungen für die globale Variable `errcode`, so dass deren Verwendung in den Dateien `solver.c` und `main.c` fehlerfrei möglich wird.

(c) Vereinbaren Sie in jeder der beiden Quelldateien `solver.c` und `main.c` eine Funktion `getSourceFilename()`, die den jeweiligen Namen der Quelldatei als Zeichenkette zurückgibt.

(d) Implementieren Sie `getSourceFilename()` zusätzlich als Präprozessor-Makro. Falls beim Übersetzen ein Makro mit dem Namen `INLINE` definiert ist, sollen die innerhalb der `printf`-Aufrufe kodierten Aufrufe von `getSourceFilename()` vom Präprozessor als Makroaufrufe erkannt werden. Andernfalls sollen sie weiterhin als Funktionsaufrufe interpretiert werden.

4 Zeiger-Operationen

Gegeben sind die Vereinbarungen:

```
int a[5] = { 1, 2, 3, 4, 5 };
int *p = a+2;
int *q = &a[5];
void *v = (void *) q;
```

Geben Sie (durch Ankreuzen) für jeden der nachfolgenden Ausdrücke an, ob dieser zulässig ist („ja“) oder nicht („nein“).

ja	nein	
☒	☐	`*a + 2`
☒	☐	`p - 1`
☐	☒	`p + q - v`
☐	☒	`*p * *q`
☐	☒	`*p * *v`
☐	☒	`p * *q`
☐	☒	`* *q`
☒	☐	`* &q`
☒	☐	`& *q`
☒	☐	`p <= q`
☐	☒	`*p <= q`
☒	☐	`(p - q) <= 3`

5 **Text kodieren/dekodieren**

Schreiben Sie eine Funktion, die Buchstaben in einem gegebenen Text verschlüsselt bzw. entschlüsselt. Der Text soll als Zeichenkette beim Aufruf an die Funktion übergeben werden. Ebenso ein ganzzahliger Wert *n* und eine Kennung, die angibt, ob der Text verschlüsselt oder entschlüsselt werden soll.
Beim Verschlüsseln ist jeder Buchstabe durch seinen *n*-ten Nachfolger im Alphabet ($n > 0$, Nachfolger von 'Z' ist 'A') zu ersetzen. Groß-/Kleinschreibung soll beibehalten werden. Zeichen, die keine Buchstaben sind, dürfen nicht verändert werden.

Die Funktion soll einen Zeiger auf die veränderte Zeichenkette zurückgeben.

Hinweis: U.u. sind folgende Funktionen aus `ctype.h` hilfreich:

```
int isalpha(int c);  /* Test, ob Buchstabe           */
int islower(int c);  /* Test, ob Kleinbuchstabe      */
int isupper(int c);  /* Test, ob Grossbuchstabe      */
int tolower(int c);  /* Umwandlung in Kleinbuchstabe */
int toupper(int c);  /* Umwandlung in Grossbuchstabe */
```

```
/* Datei: codec.h */
#ifndef CODEC_H_INCLUDED
#define CODEC_H_INCLUDED

typedef enum {encode, decode} codec_mode_t;
/* COdierer-DECodierer (public) */
extern char *codec(char *text, int n, codec_mode_t mode);

#endif /* CODEC_H_INCLUDED */
```

```
/* Datei: codec.c */

#include <ctype.h>
#include <assert.h>
#include "codec.h"

/* Codierer (private) */
static char *encoder(char *text, int n) {
  int i, k;                        /* 0 <= n <= 26  */

  for (i = 0; text[i] != '\0'; ++i) {
    if (isalpha(text[i])) {
      k = tolower(text[i])-'a';  /* 0 <= k <= 25  */
     /* verschlüsselter Buchstabe = ('a'|'A')+k+n */
     /*   = ursprünglicher Buchstabe + n          */
                /* für k+n gilt: 0 <= k+n <= 51  */
      text[i] += ((k+n) < 26) ? n : n-26;
    }
  }
  return text;
}
```

```c
/* COdierer-DECodierer (public) */
char *codec(char *text, int n, codec_mode_t mode) {
  assert(n >= 0);
  n %= 26;      /* für n gilt:    0 <=  n   <= 25  */
                /* für 26-n gilt: 0 <  26-n <= 26  */
  return encoder(text, (mode == encode) ? n : 26-n);
}
```

---

2. Lösungsvorschlag:

```c
/* Datei: codec1.h */

#ifndef CODEC1_H_INCLUDED
#define CODEC1_H_INCLUDED

typedef enum {encode, decode} codec_mode_t;

#include <assert.h>

/* COdierer-DECodierer */
#define codec(text, n, mode) ( assert((n) >= 0), \
          encoder((text), ((mode) == encode) ? (n) : -(n)) )

/* Codierer */
extern char *encoder(char *text, int n);

#endif /* CODEC1_H_INCLUDED */
```

```c
/* Datei: codec1.c */

#include <ctype.h>
#include "codec1.h"

#define int2alpha(i, c) ( (c) + (i)%26 + ((i)%26 < 0)*26 )
#define int2lower(i) int2alpha(i, 'a')
#define int2upper(i) int2alpha(i, 'A')

/* Codierer */
char *encoder(char *text, int n /* n beliebig */) {
  char *result =text;

  while (*text) {
    if (isalpha(*text)) {
      int k = tolower(*text)-'a';
      *text = islower(*text) ? int2lower(k+n) : int2upper(k+n);
    }
    text++;
  }
  return result;
}
```

6 String in Substrings zerlegen

Um einen Substring innerhalb einer gegebenen Zeichenkette zu beschreiben, reicht es aus, die Adresse des ersten Zeichens des Substrings anzugeben und sich dessen Länge zu merken. Enthält eine Variable `text` beispielsweise die Zeichenkette `"abcdefg"`, dann beschreibt das Tupel `(&text[1], 3)` die Teilzeichenkette `"bcd"`.

(a) Vereinbaren Sie einen Datentyp `substring_t` zum Speichern von Substrings. Ein Objekt dieses Typs soll nur die Werte eines Tupels enthalten, welches einen Substring spezifiziert, jedoch keine Kopie der im Substring vorliegenden Zeichen.

```
typedef struct {
  char *str;
  int len;
} substring_t;
```

(b) Schreiben Sie eine Funktion `splitString`. Diese soll eine Zeichenkette übergeben bekommen, die aus mehreren, durch das Zeichen ':' voneinander getrennten, Teilzeichenketten besteht. Die Funktion `splitString` soll für jeden Teilstring ein `substring_t`-Objekt in einem entsprechenden Feld ablegen, welches vom Aufrufer der Funktion bereitzustellen ist. Die Anzahl der abgespeicherten Substrings soll als Resultatwert der Funktion zurückzugegeben werden.

Beispiel für einen Aufruf mit Resultatwert 3:
`splitString("/usr/X11R6/lib:/lib:/usr/lib" . . .`

```
#define DELIM ':'

int splitString(const char *str, substring_t substr[])
{
  const char *s = str;
  int n = 0;

  for ( ; *str != '\0'; str++)
  {
    if (*str == DELIM)
    {
      substr[n].str = (char *) s;
      substr[n++].len = str - s;
      s = str+1;
    }
  }
  substr[n].str = (char *) s;
  substr[n++].len = str - s;

  return n;
}
```

(c) Schreiben Sie eine Funktion `printStrings`, die alle in einem Feld mit Elementen des Typs `substring_t` enthaltenen Substrings auf der Standardausgabe ausgibt.
Beispiel für die Ausgabe:

```
0. /usr/X11R6/lib
1. /lib
2. /usr/lib
```

```
#include <stdio.h>

void printStrings(const substring_t substr[], int n)
{
  int i;

  for (i = 0; i < n; i++)
  {
    printf("%d. %.*s\n", i,
      substr[i].len, substr[i].str);
  }
}
```

7 Spezielle Funktion `errormsg`

(a) Vereinbaren Sie ein statisches Feld mit Elementen vom Typ „Zeiger auf Zeichenketten“. Das Feld soll mit Zeigern auf die folgenden Fehlermeldungen initialisiert werden:

```
"Success"
"Range␣error"
"Domain␣error"
```

```
const char *errmsg[] = {
  "Success",
  "Range␣error",
  "Domain␣error"
};
```

(b) Schreiben Sie eine Funktion `errormsg`, der beim Aufruf ein **`int`**-Wert übergeben wird. Es ist davon auszugehen, dass dieser **`int`**-Wert 4 Bytes im Speicher belegt. Falls alle Bits innerhalb der beiden höherwertigen Bytes des übergebenen Wertes auf 0 gesetzt sind, ist der **`int`**-Wert als Index für das in (a) vereinbarte statische Feld aufzufassen und die unter diesem Index gespeicherte Fehlermeldung auszugeben. Andernfalls soll der **`int`**-Wert als Speicheradresse interpretiert werden, und die Fehlermeldung ausgegeben werden, die sich an dieser Speicheradresse befindet. Die Teilaufgabe (c) zeigt ein Beispiel für die Verwendung dieser Funktion.

```
#include <stdio.h>

void errormsg(int val) {
  printf("%s\n", (0xFFFF0000 & val) ?
                    (char *) val : errmsg[val]);
}
```

(c) Das folgende Programm ist ein Anwendungsbeispiel für die Funktion `errormsg`:

```
1 extern void errormsg(int);
2 #define errormsg(arg) errormsg((int) arg)
3
4 int main(void) {
5   errormsg(2);              /* Ausgabe: Domain error */
6   errormsg("Domain error");/* Ausgabe: Domain error */
7   return 0;
8 }
```

i. Wozu dient die Definition des Makros `errormsg` in Zeile 2?

ii. Wie wirkt sich das Weglassen (bzw. Auskommentieren) dieser Zeile aus?

iii. Was passiert, wenn die Zeilen 1 und 2 vertauscht werden?

i. Dieses Makro stellt sicher, dass der bei einem Aufruf an die Funktion `errormsg` übergebene Argumentwert explizit in einen **`int`**-Wert umgewandelt wird.

ii. Zeile 6 ist nun unzulässig, da an die Funktion `errormsg` kein **`int`**-Wert, sondern ein Zeiger auf ein **`char`**-Objekt übergeben wird.

iii. In diesem Fall wird die Funktionsdeklaration

```
extern void errormsg(int);
```

vom Präprozessor abgewandelt zu:

```
extern void errormsg((int) int);
```

Der Versuch, diese Zeile zu übersetzen, führt zu einem Syntax-Fehler.