**Praktikums-Aufgaben zur modularen Programmierung in C**

Informatik 2, Fakultät efi; WS 2016/17

Zu jeder Modul-Aufgabe ist eine *.c und *.h Modul-Datei (Modulname.c) und eine *.c Test-Datei (TestModulname.c mit main() Funktion) zu erstellen, die es ermöglicht alle Modulfunktionen zu testen.
Weiterhin soll jedes Modul (auch) als Bibliothek (mit lib- und .h-Datei) zur Verfügung gestellt werden. Dafür wird die Verwendung von make mit der Erstellung einer Make-Datei empfohlen.

**Wichtiger Hinweis:**
**Eine dynamische Verwaltung von Speicher muss in allen Aufgaben mit malloc() und free() durchgeführt werden; dynamische Arrays sind grundsätzlich nicht erlaubt !!!**

**Modul-Aufgabe 1:**
Rechnen mit komplexen Zahlen (complex.c, complex.h, TestComplex.c)
Erstellen Sie ein Programm das das Rechnen mit komplexen Zahlen ermöglicht.
Das Programm soll Funktionen für die „Grundrechenarten“ und die „formatierte Ausgabe“ einer komplexen Zahl bereitstellen.

a) Erstellen Sie, passend zur vorgegebenen Header-Datei complex.h, die Funktionen für die Grundrechenarten und die formatierte Ausgabe in einem Modul complex.c.
b) Erstellen Sie weiter ein Test-Programm TestComplex.c, das es ermöglicht alle Funktionen zu überprüfen, indem es jeweils 2 komplexe Zahlen einliest und das Ergebnis für alle Rechenarten ausgibt. (Alternativ kann zusätzlich auch die Rechenart (A, S, M, D) eingelesen, die gewählte Rechenart ausführt und das Ergebnis ausgegeben werden).
c) Compilieren und Linken Sie die beiden Moduldateien und führen Sie einen Test aller Modul Funktionen durch.
d) Erzeugen Sie für das Modul complex.c eine Bibliothek complex.lib / libcomplex.a. Binden Sie die Bibliothek zu dem Test-Programm und wiederholen Sie die Tests.

Hinweis: Eine komplexe Zahl x ist in ihrer algebraischen Form darstellbar als
x = a + bi, wobei x € C und a, b € R. a wird Realteil und b Imaginärteil der komplexen Zahl genannt.
Für komplexe Zahlen (hier x = a + bi und y = c + di) gelten folgende Rechenregeln:
Addition:
z = (a + bi) + (c + di) = (a + c) + (b + d)i
Subtraktion:
z = (a + bi) - (c + di) = (a - c) + (b - d)i
Multiplikation:
z = (a + bi)(c + di) = (ac - bd) + (ad + bc)i
Division (y = c + di != 0):
z = (a+bi) / (c+di) = (ac+bd) / (c^2+d^2) + (bc-ad) / (c^2+d^2) i
Hinweis: Vor einer Division muss überprüft werden, ob der Nenner gleich 0 ist.

Mögliche Testabläufe:

```
1.Zahl eingeben
      Realteil: 3
 Imaginaerteil: 4
2.Zahl eingeben
      Realteil: 5
 Imaginaerteil: 6

x = (3.000 + 4.000i)
y = (5.000 + 6.000i)

Summe:      x + y = (8.000 + 10.000i)
Differenz: x - y = (-2.000 + -2.000i)
Produkt:   x * y = (-9.000 + 38.000i)
Quotient:  x / y = (0.639 + 0.033i)
```

```
1.Zahl eingeben
      Realteil: 1.2
 Imaginaerteil: 2.3
2.Zahl eingeben
      Realteil: 3.4
 Imaginaerteil: 4.5

x = (1.200 + 2.300i)
y = (3.400 + 4.500i)

Summe:      x + y = (4.600 + 6.800i)
Differenz: x - y = (-2.200 + -2.200i)
Produkt:   x * y = (-6.270 + 13.220i)
Quotient:  x / y = (0.454 + 0.076i)
```

**Modul-Aufgabe 2:**
Rechnen mit Matrizen (matrixOp.c, matrixOp.h, TestMatrixOp.c)
Erstellen Sie ein Programm, das Matrixoperationen bzw. Funktionen zum Rechnen mit Matrizen ermöglicht.

a) Implementieren Sie die in der bereitgestellten Headerdatei matrixOp.h deklarierten Funktionen in einem Modul matrixOp.c

b) Erstellen Sie weiter ein Testprogramm TestMatrixOp.c, das alle Funktionen des Moduls aufruft und es ermöglicht diese zu überprüfen und zu testen.

c) Erzeugen Sie für das Modul matrixOp.c eine Bibliothek matrixOp.lib / libmatrixOp.a.
Binden Sie die Bibliothek zu dem Test-Programm und führen Sie einen Test aller Modul Funktionen durch.

Hinweis: Für die Tests können Matrizen im Testprogramm automatisch mit Werten, z.B. aufsteigend mit einzugebendem Anfangswert, gefüllt werden.

Mögliche Testabläufe:

```
Test verschiedener Funktionen der Bibliothek
Gewuenschte Matrizen-Groesse eingeben
 Zeilen, Spalten (> 0; z.B.: 3,4): 2,3
Matrix Elemente eingeben (F. Test nur 1.Elem.,
 weitere Elemente werden mit +1 erzeugt)
 Element in [1,1] (z.B.: 4.5): 4.5

Test Create Zero und Rand:
CreateMatrixZero: a[2,3]=
(    0.00     0.00     0.00)
(    0.00     0.00     0.00)

Info: Called srand!
CreateMatrixRand: a[2,3]=
(   44.81   -86.41   -42.44)
(  -65.48   -15.88    18.40)

Tests mit eingegebenen Werten:
CreateMatrix: a[2,3]=
(    4.50     5.50     6.50)
(    7.50     8.50     9.50)

CreateMatrix: b[2,3]=
(    4.50     5.50     6.50)
(    7.50     8.50     9.50)

c[2,3]= a + b =
(    9.00    11.00    13.00)
(   15.00    17.00    19.00)

c[2,3]= a - b =
(    0.00     0.00     0.00)
(    0.00     0.00     0.00)

c[3,2]= b^T
b[3,2]= c =
(    4.50     7.50)
(    5.50     8.50)
(    6.50     9.50)

c[2,2]= a * b =
(   92.75   142.25)
(  142.25   218.75)

det(c) = 54.00
```

```
Test verschiedener Funktionen der Bibliothek
Gewuenschte Matrizen-Groesse eingeben
 Zeilen, Spalten (> 0; z.B.: 3,4): 2,3
Matrix Elemente eingeben (F. Test nur 1.Elem.,
 weitere Elemente werden mit +1 erzeugt)
 Element in [1,1] (z.B.: 4.5): 7

Test Create Zero und Rand:
CreateMatrixZero: a[2,3]=
(    0.00     0.00     0.00)
(    0.00     0.00     0.00)

Info: Called srand!
CreateMatrixRand: a[2,3]=
(  -63.34   -43.53    -1.27)
(   23.94    57.35    49.62)

Tests mit eingegebenen Werten:
CreateMatrix: a[2,3]=
(    7.00     8.00     9.00)
(   10.00    11.00    12.00)

CreateMatrix: b[2,3]=
(    7.00     8.00     9.00)
(   10.00    11.00    12.00)

c[2,3]= a + b =
(   14.00    16.00    18.00)
(   20.00    22.00    24.00)

c[2,3]= a - b =
(    0.00     0.00     0.00)
(    0.00     0.00     0.00)

c[3,2]= b^T
b[3,2]= c =
(    7.00    10.00)
(    8.00    11.00)
(    9.00    12.00)

c[2,2]= a * b =
(  194.00   266.00)
(  266.00   365.00)

det(c) = 54.00
```

**Modul-Aufgabe 3:**
Eine grundlegende Datenstruktur sind so genannte Queues, die mit Warteschlangen vergleichbar sind. Operationen auf Queues sind:

- **put():** fügt ein Element am Ende der Warteschlange hinzu.
- **get():** entnimmt ein Element am Anfang der Warteschlange und liefert es zurück.


Erstellen Sie ein Programm, das diese beiden Operationen realisiert.

a) Implementieren Sie die in der bereitgestellten Headerdatei queue.h deklarierten Funktionen in einem Modul queue.c. Realisieren Sie dabei die Queue in Form einer verketteten Liste mit zwei Zeigern auf den Anfang und das Ende der Liste.
   put() hängt übergebene Zahl an das Ende der Liste an (mit Aktualisierung des Ende-Zeigers)
   get() liefert das Element am Anfang der Liste (mit Aktualisierung des Anfang-Zeigers)
   Fangen Sie dabei mit entsprechenden Ausgabe-Meldungen Situationen ab, bei denen get() auf eine leere Liste angewendet wird sowie put() bei fehlenden Speicherplatz.

b) Erzeugen Sie für das Modul queue.c eine Bibliothek queue.lib / libqueue.a.
   Binden Sie die Bibliothek zu dem bereitgestellten Test-Programm **josephus.c** und führen Sie folgende Tests durch:

```
Wie viele Personen: 10
Wie vielte ist auszusondern: 2
2, 4, 6, 8, 10, 3, 7, 1, 9, 5,
```

```
Wie viele Personen: 20
Wie vielte ist auszusondern: 5
5, 10, 15, 20, 6, 12, 18, 4, 13, 1, 9, 19, 11, 3, 17, 16, 2, 8, 14, 7,
```

*Das Programm simuliert das sogenannte „Josephus-Spiel“.*
Das Josephus-Spiel:
Im Jahre 67 n. Chr. wurde die galiläische Stadt Jotapata nach 47 tägiger Belagerung von Kaiser Vespasian eingenommen. Josephus, der Anführer des Widerstands, und 40 Soldaten wollten nun, um der Sklaverei zu entgehen, sich selbst umbringen; Josephus beschwor sie vergebens, davon abzulassen. Damit er wenigstens seinen Freund und sich selbst rette, schlug Josephus als Tötungsritual die decimatio (Aussonderung jedes Zehnten) vor. An welche Stelle stellte er seinen Freund und sich, um zu überleben?
Das Programm josephus ordnet n numerierte Personen an; dann wird, beginnend mit Nummer m, jede m-te Person ausgesondert. Das Programm soll die Reihenfolge der Aussonderung ausgeben. Dabei sollten Sie mit einer verketteten Liste arbeiten. Mögliche Abläufe des Programms josephus sind gezeigt.