Ingenieurinformatik (FK 03)
Übung 6

VORBEREITUNG

Bereiten Sie den Quellcode weitestgehend vor.

ÜBUNG 6A (ÜBERGABE DER ADRESSE EINES ARRAYS ALS PARAMETER)

Einführung

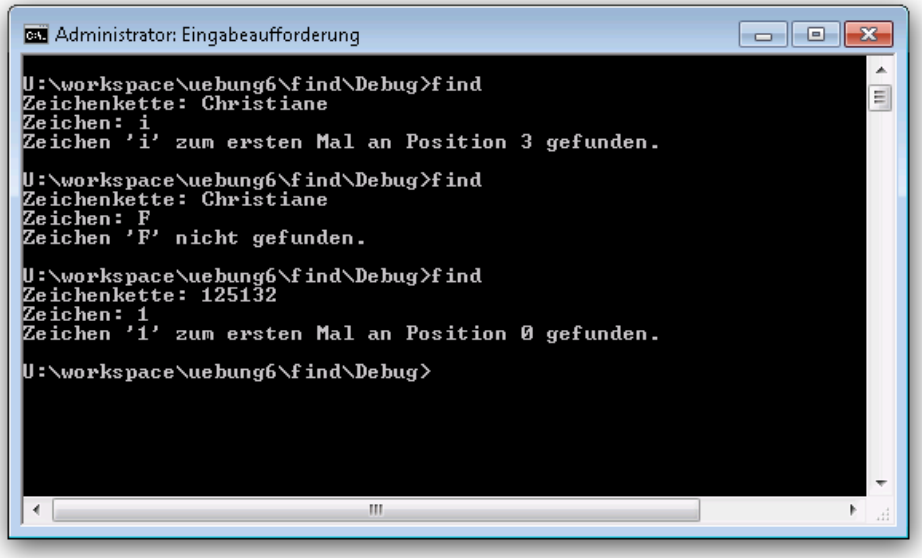
Erstellen Sie eine Funktion **`find_first`**, die als Parameter eine Zeichenkette sowie ein einzelnes Zeichen übergeben bekommt. (Übergeben Sie die Zeichenkette per Zeiger.)

Die Funktion **`find_first`** ermittelt diejenige Position innerhalb der Zeichenkette, an der das angegebene Zeichen zum ersten Mal vorkommt.
Als Rückgabewert gibt **`find_first`** die gefundene Position zurück oder -1, falls das angegebene Zeichen gar nicht in der Zeichenkette vorkommt.

Ziel

Zum Testen der Funktion soll ein geeignetes Hauptprogramm erstellt werden, das

- zuerst vom Benutzer einen String (max. 50 Zeichen) erfragt,
- dieser soll in ein geeignetes **`char`**-Array abgelegt werden, das in **`main`** definiert wird,
- danach das zu suchenden Zeichen vom Benutzer erfragt,
- die Funktion **`find_first`** mit den Eingaben aufruft und
- das Ergebnis entsprechend den nachfolgenden Screenshots ausgibt.



Durchführung

- Erstellen Sie ein **C-Projekt** mit dem Namen **`find`** unter dem Verzeichnis **`U:\workspace`**
- Erstellen Sie die Quelldatei **`find_main.c`** und realisieren Sie die Aufgabe gemäß den getroffenen Vorgaben.
- Verwenden Sie zur Eingabe des Strings die Funktion **`scanf`** mit dem Formatspezifizier **`"%50s"`**.

- Verwenden Sie zur Eingabe des Suchzeichens die Funktion **`scanf`** mit dem Formatspezifizier **`"%c"`**.
- Da sich nach dem ersten **`scanf`**-Aufruf (Eingabe des Strings) noch der Zeilenvorschub (**`'\n'`**) im Tastaturpuffer befindet, würde der zweite **`scanf`**-Aufruf nicht auf ein Zeichen von der Tastatur warten, sondern das Zeichen **`'\n'`** aus dem Puffer abholen.
  Zum Leeren des Tastaturpuffers verwenden Sie **vor** dem zweiten **`scanf`**-Aufruf, folgende Anweisungszeile:
  ```
  while (getchar()!='\n');
  ```
- Rufen Sie das Programm auf und testen Sie Ihre Funktion **`find_first`**.

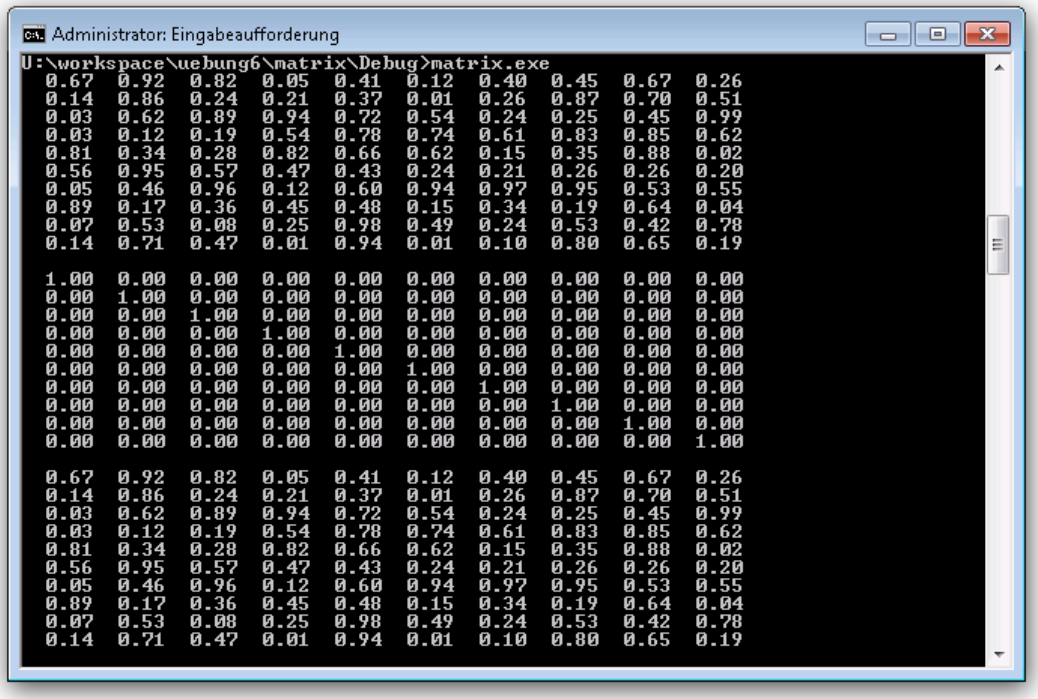
ÜBUNG 6B (MATRIZEN – 2 DIM. ARRAYS)

Ziel

Erstellen Sie auf Basis des vorgegebenen Quellcodes **`mat_mult.c`** ein C-Programm zur Multiplikation von 10x10-Matrizen mithilfe globaler zweidimensionaler Arrays.
Testen Sie Ihr Programm, indem Sie eine mit Zufallszahlen gefüllte 10x10-Matrix mit einer 10x10-Einheitsmatrix multiplizieren.

Durchführung

- Schließen Sie ggf. bereits geöffnete Projekte und erstellen Sie ein neues **C-Projekt** mit dem Namen **`matrix`**.
- Kopieren Sie nun die bereitgestellte Datei **`mat_mult.c`**, welche zwei 3x3 Matrizen miteinander multipliziert in das Verzeichnis **`U:\workspace\matrix`** mithilfe des Windows-Explorer.
- Fügen Sie die Datei dem Projekt hinzu und entfernen die bereitgestellte Datei **`main.c`**.
- Erstellen Sie das Programm **`matrix`** und starten es.
- **Frage B1:** Welches Ergebnis erwarten Sie?
  Überprüfen Sie das Ergebnis auf Richtigkeit?
- Ändern Sie das Programm so ab, dass mit 10x10- Matrizen anstelle der bisherigen 3x3-Matrizen gearbeitet wird.
- Erstellen Sie zwei Funktionen, um die Elemente des zwei-dimensionalen Arrays **`a`** und **`b`** wie folgt zu setzen:
  - Erstellen Sie eine Funktion **`void fill_a(void)`**, welche die Matrix **`a`** mit Zufallszahlen zwischen 0 und 1 füllt.
    Verwenden Sie zwei **`for`**-Schleifen.
    Zur Erzeugung einer Zufallszahl steht Ihnen die Funktion **`rand`** der Standardbibliothek zur Verfügung.
  - Erstellen Sie eine Funktion **`void fill_b(void)`**, welche die Matrix **`b`** als Einheitsmatrix belegt (alle Elemente auf der Hauptdiagonalen = 1, alle anderen Elemente = 0).
    Verwenden Sie zwei **`for`**-Schleifen.
- Entfernen Sie in **`main`** die Kommentarzeichen für die *Initialisierung des Zufallszahlengenerators* und für das *Setzen und der Ausgabe der Matrizen*.
- Erstellen Sie das modifizierte Programm **`matrix`**, starten es und überprüfen das Ergebnis.
  Das nachfolgende Bild zeigt einen exemplarischen Aufruf des Programms:



OPTIONALE ZUSATZÜBUNGEN 6C (ÜBERGABE VON ADRESSEN ALS PARAMETER)

Einführung

C-Funktionen können nur **einen** oder **keinen** Rückgabewert an den Aufrufer liefern.
Auch wird der Typ des Rückgabewertes zur Übersetzungszeit festgelegt, d. h. man kann nicht einmal z. B. einen **`int`**-Wert und durch einen weiteren Aufruf einen **`double`**-Wert an den Aufrufer zurückgeben.

Soll eine Funktion mehrere Variablen ändern, dann ist dies nicht mehr mithilfe der Rückgabe eines elementaren Datentyps möglich.
Eine Möglichkeit dies zu umgehen ist die Definition globale Variable, die somit dem Aufrufer und der aufgerufenen Funktion zur Verfügung stehen.

```c
#include <stdio.h>

/* globale Variablen */
double real, imag;

int addComplex(double re, double im)
{
	int retval=0;
	/* Speichere das Ergebnis der Addition (real+re)+i*(imag+im)
	 * in den globalen Variablen real und imag.
	 */
	real+=re; imag+=im;
	if (imag!=0)
		retval=1;
	return retval;
}
/* Hauptprogramm */
int main()
{
	real=1; imag=2;	addComplex(2, 1);
	printf("Ergebnis: %.2f + %.2fi\n", real, imag);

	real=3; imag=2;	addComplex(real, imag);
	printf("Ergebnis: %.2f + %.2fi\n", real, imag);
	return 0;
}
```

Auch wenn diese Möglichkeit leicht verständlich ist, besitzt sie einige Nachteile, die in der Softwareentwicklung vermieden werden sollten (z. B. Festlegung mit welchen Variablennamen gearbeitet wird →Bruch der Modularisierung, Probleme bei der mehrfach quasiparallelen Ausführung einer Funktion, ...).
Eine Alternative bietet die Übergabe der Adresse einer Variablen des Aufrufers.

```
#include <stdio.h>

int addComplex(double *pReal, double *pImag,
               double re, double im)
{
       int retval=0;
       /* Speichere das Ergebnis der Addition
        * (*pReal) + re + i*( (*pImag)+im )
        * in die Variablen des Aufrufers, deren Speicheradressen in pReal und pImag
        * uebergeben wurde.
        */
       *pReal = *pReal + re; *pImag = *pImag + im;
       if (*pImag!=0)
              retval=1;
       return retval;
}

int main()
{
       double re1=1, im1=2;
       double re2=3, im2=2;

       /* Das Ergebnis der nachfolgenden Anweisung wird in re1 und im1 abgelegt */
       addComplex(&re1, &im1, 2, 1);
       /* Das Ergebnis wird nun in re2 und im2 abgelegt */
       addComplex(&re2, &im2, re2, im2);
       printf("Ergebnis: %.2f + %.2fi\n", re1, im1);
       printf("Ergebnis: %.2f + %.2fi\n", re2, im2);
       return 0;
}
```

Ziel

Erweitern Sie beide Beispiele um weitere Funktionen zum Rechnen mit komplexen Zahlen und ergänzen Sie die Funktion **`main`** um Aufrufe, die die Funktionen testen:

- **`int subComplex(double re, double im);`** **`/* Beispiel 1 */`**
  `int subComplex(double *pReal, double *pImag, double re,`
  `double im);` **`/* Beispiel 2 */`**
  Subtrahiert zwei komplexe Zahlen voneinander.

- **`int normal(double betrag, double phase);`** **`/* Beispiel 1 */`**
  `int normal(double *pReal, double *pImag, double betrag,`
  `double phase);` **`/* Beispiel 2 */`**
  Ermittelt aus der übergebenen komplexen Zahl in Polarform die entsprechende Zahl in Normalform.

- **`int polar(double re, double im);`** **`/* Beispiel 1 */`**
  `int polar(double *pBetrag, double *pPhase, double re,`
  `double im);` **`/* Beispiel 2 */`**
  Ermittelt aus der übergebenen komplexen Zahl in Normalform die entsprechende Zahl in Polarform.
  Verwenden Sie zur Berechnung von Betrag und Phase (in Bogenmaß) die Standard-C-Funktionen **`sqrt`** und **`atan2`**.
  Definieren Sie für **Beispiel 1** zwei neue globale Variablen namens **`betrag`** und **`phase`**.

Alle Funktionen liefern die C-Entsprechung für **`true`**, wenn das Resultat eine komplexe Zahl ist.