Übungspaket 26

Der Datentyp `struct`

Übungsziele:

1. Organisation von `structs` im Arbeitsspeicher
2. Problemangepasste Verwendung von `structs`.

Skript:

Kapitel: 53

Semester:

Wintersemester 2020/21

Betreuer:

Thomas, Tim und Ralf

Synopsis:

Strukturen (`structs`) sind nach den Arrays die zweite Form komplexer Datenstrukturen. Ein wesentliches Element von Strukturen ist, dass sie *unterschiedliche* Variablen zu einem neuen (komplexen) Datentyp zusammenfügen können. Dieser neue Datentyp kann anschließend wie jeder andere verwendet werden. Zu einem sind `structs` ein hervorragendes Strukturierungsmittel. Zum anderen sind sie aufgrund ihrer Eigenschaft des Zusammenfügens *unterschiedlicher* Datentypen eine wesentliche Voraussetzung dynamischer Datenstrukturen (Übungspakete 31 bis 33). Insofern ist dieses Übungspaket von großer Wichtigkeit.

Teil I: Stoffwiederholung

Aufgabe 1: Strukturen vs. Arrays

Strukturen und Arrays gehören zu den komplexen Datentypen. Für den Programmieranfänger ist es anfangs oft schwer, diese beiden Strukturierungsmethoden auseinander zu halten, obwohl dies für die weitere Arbeit von besonderer Bedeutung ist. Erkläre in eigenen Worten, was `struct`s (Strukturen) sind und was der Sinn dahinter ist.

Eine Struktur (`struct`) ist in erster Linie ein Strukturierungselement, dass die Datenorganisation problemadäquat und damit lesbarer machen soll. In der Regel sorgt eine Struktur dafür, dass die Daten im Arbeitsspeicher so angelegt werden, dass zusammengehörende Elemente dicht beieinander liegen. Das typische Beispiel sind alle Angaben zu einer Person. Ein wesentliches Merkmal von Strukturen ist, dass sie Datenelemente *unterschiedlichen* Typs zusammenfassen können.

Strukturen und Arrays unterscheiden sich vor allem in zweierlei Hinsicht. Stelle diese in der folgenden Tabelle einander gegenüber.

Aspekt	Arrays	Strukturen
Datentypen der Elemente bzw. der Komponenten	In einem Array haben *immer* alle Elemente den selben Typ; es kann keine Unterschiede geben.	Ein `struct` kann Elemente verschiedener Datentypen zu einem neuen Typ zusammenfassen.
Zugriff auf die einzelnen Elemente bzw. Komponenten	Auf die einzelnen Elemente eines Arrays greift man über seinen Namen und einem Index zu, der in eckigen Klammern steht. Beispiel: `a[ i ]`	Auf die einzelnen Komponenten eines `struct`s kann man zugreifen, in dem man beide Namen mittels eines Punktes verbindet. Beispiel: `struktur.komponente`

Lassen sich Arrays und `struct`s kombinieren | Ja! Sogar beliebig verschachtelt.

Aufgabe 2: Beispiele

Zeige anhand dreier Beispiele, wie `struct`s definiert werden:

```
1 struct cpx_nr { double re, im; } n1, n2;  // 2 complex numbers
2
3 struct ic { char c; int i; };         // just a char and an int
4
5 struct cstr {int len; char str[ 20 ] };    // a complex string
```

Zeige anhand einiger Beispiele, wie man auf die einzelnen Komponenten eines `structs` zugreift. Berücksichtige dabei auch mindestens ein Array:

```
1 struct cpx_nr { double re, im; };          // data type complex
2 struct cpx_nr n1, coefs[ 10 ];          // 1 simple, 1 array var
3
4 n1.re = 1.0; n1.im = -1.0;             // setting the components
5 coefs[ 1 ].im = sin( -0.5 );            // access array elements
6 coefs[ 3 ].re = n1.re * coefs[ 1 ].im;           // more complex
7 coefs[ 2 ] = n1;                          // copying all at once
8 coefs[ 0 ] = coefs[ 2 ];                  // copying all at once
```

Natürlich müssen wir auch Zeiger auf `structs` können. Nehmen wir an, wir haben einen Zeiger `p` auf einen `struct s`, in dem sich eine Komponente `i` vom Typ `int` befindet. Auf welche beiden Arten kann man auf die Komponente `i` desjenigen `structs` `s` zugreifen, auf das `p` zeigt?

1. `(*p).i = 4711`
2. `p->i = 4711`

Zeichne für folgende `struct`-Definition ein Speicherbild:

```
1 struct two_strings { char *first, *second; };
```

Struktur	Komponente	Wert
	`char *second:`	
`two_strings`	`char *first :`	

Wie viele Bytes belegt ein derartiges `struct`, wenn ein Zeiger vier Bytes belegt? 8

Nun nehmen wir folgende Variablendefinition an:

```
1 struct two_strings example = { "Enrico", "Johanna" };
```

Wie viele Bytes belegt eine derartige Variable? 8 bzw. 23 Bytes.

Anmerkung: Die Antwort hängt davon ab, ob man die Konstanten mitzählt. Korrekt ist eigentlich 8, da die Variable aus 2 Zeigern besteht; die Namen gehören nicht dazu.

Zeichne hierfür ein Speicherbildchen:

Adresse	Var.	Komponente	Wert
0xFE24		`char *second:`	0xF838
0xFE20	`example`	`char *first :`	0xF830

Adresse	Wert
0xF83C	`'n' 'n' 'a' '\0'`
0xF838	`'J' 'o' 'h' 'a'`
0xF834	`'c' 'o' '\0'`
0xF830	`'E' 'n' 'r' 'i'`

Aufgabe 3: Rekursive `struct`-Definitionen

Zur Erinnerung: *Rekursion* bedeutet, sich selbst wieder aufzurufen. Das haben wir bereits bei Funktionen kennengelernt und dort auch deren Vorzüge gesehen. Ein wichtiger Aspekt bei rekursiven Funktionsaufrufen ist, dass man sie irgendwann terminieren muss.

Nun zu den `structs`: Nehmen wir an, wir hätten folgende Definition:

```
1 struct rekursion {
2                     int i;
3                     struct rekursion noch_was;
4                   };
```

Erkläre mit eigenen Worten, weshalb eine derartige rekursive `struct`-Definition in C *nicht* möglich ist:

In oben gegebenem Beispiel beinhaltet das `struct rekursion` sich in sich selbst. Dies würde aber zu einer Endlosrekursion führen, wodurch der Speicherbedarf für eine derartige Struktur unendlich wäre. Die Programmiersprache C bietet keine Möglichkeit, die Verschachtelungstiefe irgendwie zu begrenzen. Dies entspräche den russischen Matrjoschka Puppen, wenn immer wieder eine neue Puppe um die bestehenden herum platziert würde.

Was wäre aber, wenn wir folgende Definition hätten?

```
1 struct rekursion {
2                     int i;
3                     struct rekursion *noch_mehr;
4                   };
```

Von welchem Datentyp ist die Komponente `i`? int

Von welchem Datentyp ist die Komponente `noch_mehr`? struct rekursion *
also ein *Zeiger* auf sich selbst

Dies ist in der Tat in C erlaubt. Wie viele Bytes belegt eine derartige Struktur, wenn ein `int` und ein Zeiger jeweils vier Bytes brauchen (`sizeof(struct rekursion)`)? 8 Bytes

Diskutiere mit den Kommilitonen bei einem Kaffee, Bier oder sonstwas, was man damit machen könnte.

Eine mögliche Antwort wäre: Man könnte mehrere von diesen Strukturen „aneinanderhängen“, wenn man nur mehrere davon von der CPU bekäme. Dazu werden wir in Kapitel 69 und Übungspaket 29 kommen.

Teil II: Quiz

So ein „übliches“ Quiz ist uns zu diesem Thema nicht eingefallen, da die Sachverhalte schlicht zu einfach sind. Daher beschäftigen wir uns diesmal mit Typen und Werten im Rahmen von `structs`. Dies ist insbesondere eine sehr gute Vorbereitung für die dynamischen Datenstrukturen, die wir in den Übungspaketen 31 bis 33 behandeln werden.

Aufgabe 1: `structs`, Typen, Zeiger, Werte

Nehmen wir an, wir haben folgendes C-Programm:

```
 1  struct combo {
 2            char c;
 3            int  a[ 2 ];
 4          };
 5
 6  int main( int argc, char **argv )
 7      {
 8          struct combo test;
 9          struct combo a[ 2 ];
10          struct combo *ptr;
11          test.a[ 0 ] = 4711;
12          test.a[ 1 ] = 815;
13          ptr = & test; ptr->c = 'x';
14      }
```

Dann können wir am Ende von Programmzeile `10` folgendes Speicherbildchen erstellen, in dem wie immer alle Adressen frei erfunden sind und wir davon ausgehen, dass Variablen vom Typ `int` sowie Zeiger immer vier Bytes belegen.

Adresse	Var.	Komponente		Wert
0xFE2C		int a[1]	:	815
0xFE28		int a[0]	:	4711
0xFE24	test	char c	:	'x'
0xFE20	ptr		:	0xFE24

Adresse	Var.	Komponente		Wert
0xFE44		int a[1]	:	24
0xFE40		int a[0]	:	-273
0xFE3C	a[1]	char c	:	'a'
0xFE38		int a[1]	:	24
0xFE34		int a[0]	:	-273
0xFE30	a[0]	char c	:	'a'

Ergänze zunächst die Effekte der Programmzeilen `11` bis `13` im obigen Speicherbildchen.

Vervollständige nun die folgende Tabelle. Die Ausdrücke werden im Verlaufe des Quiz immer schwieriger. Im Einzelfalle lohnt es sich, entweder ein kurzes Testprogramm zu scheiben und/oder mit den Betreuern zu diskutieren.

Ausdruck	Type	Wert	Anmerkung
`test`	`struct combo`	`----`	Die Struktur ab `0xFE24`
`sizeof( test )`	`int`	`12`	
`& test`	`struct combo *`	`0xFE24`	
`ptr`	`struct combo *`	`0xFE24`	
`sizeof( ptr )`	`int`	`4`	
`*ptr`	`struct combo`	`----`	Dies ist die Structur `test`
`sizeof( *ptr )`	`int`	`12`	Da `*ptr` die gesamte Struktur ist
`test.c`	`char`	`'x'`	
`ptr->a[0]`	`int`	`4711`	
`ptr->c`	`char`	`'x'`	
`& ptr`	`struct combo **`	`0xFE20`	
`test.a[0]`	`int`	`4711`	
`&(test.a[0])`	`int *`	`0xFE28`	
`sizeof( a )`	`int`	`24`	
`&(a[ 1 ].a[ 0 ])`	`int *`	`0xFE40`	
`*(a + 1)`	`struct combo`	`----`	Identisch mit `a[ 1 ]` (ab `0xFE3C`)

Die folgenden Quiz-Fragen sind sehr schwer!

Ausdruck	Type	Wert	Anmerkung
`sizeof( test.a )`	`int`	`8`	Hier handelt es sich tatsächlich um das ganze Array `a`
`test.a`	`int [2]`	`0xFE28`	Hier handelt es sich tatsächlich um das ganze Array `a`
`a`	`struct combo []`	`0xFE30`	Das ist das Array bestehend aus zwei Elementen vom Typ `struct combo`

Betrachte noch die folgenden Anweisungen. Trage die Auswirkungen der einzelnen Anweisung in das Speicherbildchen der vorherigen Seite ein.

```
1  a[ 0 ].c = 'a';
2  (*(a + 0)).a[ 0 ] = -273;
3  (a + 0)->a[ 1 ] = 24;
4  a[ 1 ] = a[ 0 ];
```

Teil III: Fehlersuche

Aufgabe 1: Definition und Verwendung von `structs`

Nach der Vorlesung hat Dr. Strucki versucht, ein paar `structs` zu programmieren. Offensichtlich benötigt er eure Hilfe, da ihm nicht alles klar geworden ist.

```
 1 struct cpx double re, im;                    // a complex number
 2 struct ivc { double len; cpx cx_nr; };              // plus len
 3
 4 int main( int argc, char **argv )
 5     {
 6         struct cpx cx, *xp1, *xp2;
 7         struct ivc vc, *v_p;
 8         struct ivc i_tab[ 2 ];
 9
10         cx.re = 3.0; im = -4.0;
11         vc.cx_nr = cx; vc.len = 5.0;
12         vc.re = 3.0; vc.cx_nr.im = 4.0;
13
14         xp1 = & cx; xp2 = vc.cx_nr;
15         xp1->re = 6.0; xp1.im = -8.0;
16         *xp2 = *xp1; vc.len = 10.0;
17
18         cx *= 2;
19         vc.cx_nr += cx;
20
21         (*(i_tab + 0)) = vc;
22         (i_tab + 1)->cx_nr = cx;
23     }
```

Zeile	Fehler	Erläuterung	Korrektur
1	`{}` fehlen	Bei einem `struct` *müssen* die Komponenten innerhalb `{}` stehen.	`{double ... };`
2	„`struct`“ fehlt	Bei der Verwendung von Strukturen muss das Wort `struct` dastehen.	`struct cpx`
10	`cx.` fehlt	Beim *Zugriff* auf die Komponenten muss immer auch die eigentliche Struktur angegeben werden.	`cx.im`
11		Hier ist alles richtig ;-)	

Zeile	Fehler	Erläuterung	Korrektur
12	`cx_nr.` fehlt	Hier fehlt ein Teil der kompletten Namensgebung: Beim Zugriff auf eine Komponente müssen immer alle Teile angegeben werden.	`vc.cx_nr.re`
14	`&` fehlt	`xp2` ist ein Zeiger. Entsprechend muss rechts auch die Adresse gebildet werden.	`& vc.`
15	`.` falsch	`xp1` ist ein Zeiger. Entsprechend greift man auf die Komponenten entweder mittels `->` oder `*(...).` zu.	`xp1->`
18	`cx *= 2`	Schön gedacht, aber die einzelnen Komponenten muss man schon einzeln verändern.	`cx.re *= 2 ...`
19	`+=`	dito.	
21		Sieht komisch aus, aber hier ist alles richtig.	
22		Sieht komisch aus, aber hier ist alles richtig.	

Programm mit Korrekturen:

```
 1 struct cpx { double re, im; };              // a complex number
 2 struct ivc { double len; struct cpx cx_nr; };      // plus len
 3
 4 int main( int argc, char **argv )
 5     {
 6         struct cpx cx, *xp1, *xp2;
 7         struct ivc vc, *v_p;
 8         struct ivc i_tab[ 2 ];
 9
10         cx.re = 3.0; cx.im = -4.0;
11         vc.cx_nr = cx; vc.len = 5.0;
12         vc.cx_nr.re = 3.0; vc.cx_nr.im = 4.0;
13
14         xp1 = & cx; xp2 = & vc.cx_nr;
15         xp1->re = 6.0; xp1->im =-8.0;
16         *xp2 = *xp1; vc.len = 10.0;
17
18         cx.re *= 2; cx.im *= 2;
19         vc.cx_nr.re += cx.re; vc.cx_nr.im += cx.im;
20
21         (*(i_tab + 0)) = vc;
22         (i_tab + 1)->cx_nr = cx;
23     }
```

Teil IV: Anwendungen

Aufgabe 1: Eine einfache Personenverwaltung

1. Aufgabenstellung

Gegenstand dieser Übungsaufgabe ist die Entwicklung eines kleinen Programms, das für uns eine Namensliste „verwaltet“. Damit ihr euch auf das Einüben von `structs` konzentrieren könnt, haben wir diese eigentlich doch recht komplexe Aufgabe für euch wie folgt stark vereinfacht:

1. Die Angaben zu einer Person bestehen lediglich aus Name und Alter. Desweiteren legen wir fest, dass ein Name nur aus *genau einem* Zeichen besteht.
2. Alle Personen sollen in einer Tabelle verwaltet werden. Die Zahl der Personen sowie die Daten stehen von Anfang an fest, können also direkt in das Programm integriert werden.
3. Im Rahmen unserer hypothetischen Anwendung muss unser Programm drei Dinge können:
   (a) Sortieren der Personentabelle aufsteigend nach dem Namen.
   (b) Sortieren der Personentabelle aufsteigend nach dem Alter.
   (c) Drucken der vorliegenden Tabelle.
4. Ferner benötigen wir ein Hauptprogramm, dass uns die Funktionalität gemäß obiger Beschreibung bestätigt.

2. Vorüberlegungen und Hinweise

Um euch ein wenig zu helfen, haben die Doktoranden mal ein bisschen laut nachgedacht und das Gesagte protokolliert. Dabei kamen folgende Ideen auf:

1. Wir sollen also Personen verwalten, die einen Namen vom Typ `Zeichen` sowie ein Alter vom Typ `Ganzzahl` haben. Damit die Daten beim späteren Sortieren nicht durcheinander geraten, sollten wir hierfür am besten eine Struktur definieren, in der alle Angaben Platz haben.
2. Wir brauchen eine Funktion, die uns die Tabelle auf dem Bildschirm ausgibt. Hier reicht eine Funktion, da es ihr ja egal ist, ob die Tabelle sortiert ist oder nicht.
3. Da wir die Personentabelle mal nach dem Namen und mal nach dem Alter sortieren sollen, wäre hier jeweils eine entsprechende Funktion sinnvoll, die die

Sortierung nach dem jeweiligen Kriterium vornimmt. Macht also zwei unterschiedliche Sortierfunktionen.

4. Für die Sortierfunktion können wir uns am `Bubble-Sort`-Algorithmus orientieren, den wir bereits in Übungspaket 15 hatten. Nur müssen wir dann noch einige Variablen und Tauschoperationen anpassen. Aber das haben wir ja schon im ersten Teil dieses Übungspaketes wiederholt.

5. Schließlich benötigen wir noch ein Hauptprogramm. Am Anfang basteln wir uns einfach alles zusammen, dann geben wir die Daten zur Kontrolle einmal aus, sortieren sie nach den Namen, geben sie zur Kontrolle aus, sortieren sie nach dem Alter und geben sie noch ein letztes Mal aus.

6. Jetzt sollte eigentich alles klar sein, sodass wir mit der Arbeit anfangen können.

3. Pflichtenheft

Aufgabe	: Programm zur Verwaltung von Personen, die einen Namen und ein Alter haben, Sortiermöglichkeiten nach jeweils einem der beiden Angaben.
Eingabe	: keine Eingaben, da direkte Kodierung.
Ausgabe	: Tabelle in unsortierter sowie in nach Namen bzw. Alter sortierten Form.
Sonderfälle	: keine.

Funktionsköpfe	: `prt_tab( FILE *fp, struct person *tab, int size )`
	`sort_name( struct person *tab, int size )`
	`sort_age( struct person *tab, int size )`

4. Implementierung

Da wir bereits alle benötigten Algorithmen in früheren Übungspaketen eingehend behandelt haben, können wir direkt mit der Kodierung anfangen. Falls dennoch Fragen sein sollten, einfach die Betreuer konsultieren.

5. Kodierung

Definition des `structs`:

```
1 #include <stdio.h>
2
3 struct person           // struct for a person
4         {
5            int  age;
6            char name;
7         };
```

Eine Funktion zum Drucken der Tabelle:

```
 8 int prt_tab( struct person *tab, int size )
 9     {
10        int i;
11        if ( tab )
12           for( i = 0; i < size; i++ )
13              printf("Person %3d: Name: %c Alter: %d\n",
14                     i, tab[ i ].name, tab[ i ].age );
15        else printf( "prt_tab: Tabelle vergessen\n" );
16        return tab != 0;
17     }
```

Eine Funktion zum Vertauschen zweier Elemente:

```
18 // this function makes life easier, we need it twice
19 void p_swap( struct person *p1, struct person *p2 )
20      {
21         struct person tmp;
22         tmp = *p1; *p1 = *p2; *p2 = tmp;
23      }
```

Eine Funktion zum Sortieren nach den Namen:

```
24 int sort_name( struct person *tab, int size )
25     {
26        int i, j;
27        if ( tab )
28           for( i = 0; i < size - 1; i++ )
29              for( j = 0; j < size - i - 1; j++ )
30              {
31                 if ( tab[ j ].name > tab[ j+1 ].name )
32                    p_swap( & tab[ j ], & tab[ j + 1 ] );
33                    // this time array notation...
34              }
35        else printf( "sort_name: Tabelle vergessen\n" );
36        return tab != 0;
37     }
```

Eine Funktion zum Sortieren nach dem Alter:

```
38 int sort_age( struct person *tab, int size )
39     {
40        int i, j;
41        if ( tab )
42           for( i = 0; i < size - 1; i++ )
43              for( j = 0; j < size - i - 1; j++ )
44              {
45                 if ( tab[ j ].age > tab[ j + 1 ].age )
46                    p_swap( tab + j, tab + j + 1 );
47                    // and now pointer arithmetic
48              }
49        else printf( "sort_age: Tabelle vergessen\n" );
50        return tab != 0;
51     }
```

Schließlich das Hauptprogramm:

```
52 int main(int argc, char** argv)
53     {
54        struct person ptab[] = {
55           {4, 'g'}, {12, 'm'}, {9, '9'}, {45, 'k'},
56           {1, 'c'}, {1234647675, 'b'}, {-9, 'q'},
57           {31, 'd'}, {31, 'l'}, {22, 'o'}};
58        #define PTAB_SIZE   (sizeof(ptab)/sizeof(ptab[0]))
59
60        prt_tab( ptab, PTAB_SIZE );
61        sort_name( ptab, PTAB_SIZE );
62        printf("------------------\n");
63        prt_tab( ptab, PTAB_SIZE );
64        sort_age( ptab, PTAB_SIZE );
65        printf("------------------\n");
66        prt_tab( ptab, PTAB_SIZE );
67
68        return 0;
69     }
```

Aufgabe 2: Einfache Verwaltung von Personennamen

1. Aufgabenstellung

Diese Aufgabe ist so ähnlich wie die Vorherige. Nur wollen wir diesmal, dass die Tabelle Personen verwaltet, die sowohl einen richtigen Vor- als auch einen richtigen Nachnamen haben. Beispiel: `"Herr Andrea Neutrum"`. Die Tabelle soll diesmal aber nicht sortiert werden. Vielmehr wollen wir nach Einträgen bezüglich eines Vor- oder Nachnamens suchen können und ggf. den ganzen Personeneintrag ausgeben. Wie schon in der vorherigen Aufgabe gelten wieder folgende Randbedingungen:

1. Wir benötigen eine Tablle, in der alle Personen enthalten sind. Sowohl die Tabellengröße als auch die Namen der zu verwaltenden Personen stehen von Anfang an fest, sodass diese statisch in die Tabelle eingetragen werden können.
2. Wir benötigen eine Funktion zum Drucken, zwei Suchfunktionen (Suchen nach Vor- bzw. Nachname) und ein Hauptprogramm zur Demonstration der Funktionsfähigkeit.

2. Vorüberlegungen und Hinweise

Hier wieder ein paar Hilfestellungen unserer betreuenden Doktoranden:

1. Nach der vorherigen Übung ist diese ja eigentlich schon zu einfach. Wir benötigen lediglich eine Struktur, in der zwei Namen vom Typ `char *` Platz haben.
2. Die vier benötigten Funktionen stehen schon in der Aufgabenstellung. Auch das Finden von Tabelleneinträgen haben wir bereits in Übungspaket 15 eingeübt. Das Hauptprogramm zum Funktionstest können wir wieder genau so wie in der vorherigen Übung aufbauen: Daten eintragen, Suchen, Drucken, etc.
3. Das Suchen erledigen wir am besten in getrennten Funktionen und geben einen entsprechenden Index zurück. Dieser nimmt einen speziellen Wert an, wenn der Eintrag nicht gefunden wurde.

3. Pflichtenheft

Aufgabe	: Programm zur Verwaltung von Personen, die einen richtigen Vor- und Nachnamen haben. Suchmöglichkeiten nach jeweils einer der beiden Angaben.
Eingabe	: keine Eingaben, da direkte Kodierung.
Ausgabe	: Tabelle in unsortierter Form, gefundene Personen.
Sonderfälle	: keine.

Funktionsköpfe	: `prt_tab( FILE *fp, struct person *tab, int size )`
	`find_first(char * name, struct person *tab, int size )`
	`find_last( char * name, struct person *tab, int size )`

4. Implementierung

Aufgrund der vielen Vorarbeiten können wir direkt mit der Kodierung beginnen. Falls dennoch Fragen sein sollten, einfach direkt die Betreuer konsultieren.

5. Kodierung

Definition des `structs`:

```
 1 #include <stdio.h>
 2
 3 struct person            // struct for a person
 4        {
 5           char *first, *last;
 6        };
```

Zwei Funktionen zum Drucken:

```
 7 void prt_person( FILE *fp, struct person ps )
 8      {
 9         fprintf( fp,"Person: %s %s\n",ps.first,ps.last );
10      }
11
12 void prt_notfound( FILE *fp, char *name )
13      {
14         fprintf( fp, "%s: nicht vorhanden\n", name );
15      }
```

Zwei Such-Funktionen:

```
16 int find_first( char * name,struct person *tab,int size )
17     {
18        int i;
19        for( i = 0; i < size; i++ )
20           if ( ! strcmp( name, tab[ i ].first ) )
21              return i;
22        return -1;
23     }
24
25 int find_last( char * name, struct person *tab,int size )
26     {
27        int i;
28        for( i = 0; i < size; i++ )
29           if ( ! strcmp( name, tab[ i ].last ) )
30              return i;
31        return -1;
32     }
```

Schließlich das Hauptprogramm:

```
33 int main(int argc, char** argv)
34     {
35        int i;
36        struct person ptab[] = {
37           { "Enrico", "Heinrich" }, { "Ralf", "Joost" },
38           { "Matthias","Hinkfoth" },{ "Rene","Romann" },
39           { "Ralf", "Warmuth" },{ "Ralf", "Salomon" } };
40        #define PTAB_SIZE   (sizeof(ptab)/sizeof(ptab[0]))
41
42        // test output
43        for( i = 0; i < PTAB_SIZE; i++ )
44           prt_person( stdout, ptab[ i ] );
45        printf( "---------------\n" );
46
47        i = find_first( "Matthias", ptab, PTAB_SIZE );
48        if ( i != -1 )
49           prt_person( stdout, ptab[ i ] );
50        else prt_notfound( stdout, "Matthias" );
51
52        i = find_last( "Heinrich", ptab, PTAB_SIZE );
53        if ( i != -1 )
54           prt_person( stdout, ptab[ i ] );
55        else prt_notfound( stdout, "Heinrich" );
56
57        i = find_first( "Superman", ptab, PTAB_SIZE );
58        if ( i != -1 )
59           prt_person( stdout, ptab[ i ] );
60        else prt_notfound( stdout, "Superman" );
61
62        return 0;
63     }
```