

Kapitel 14

Datenstrukturen

Datenbasis für die Beispiele diese Abschnitts

Auf den Internetseiten des Deutschen Fussball-Bundes findet man eine Tabelle mit der Bilanz aller Fußballspiele der deutschen Nationalmannschaft, die ich zur weiteren Verarbeitung in eine Textdatei (Laenderspiele.txt) geschrieben haben:

Bilanz							
Land	Spiele	gew.	unent.	verl.	Tore	Erstes Spiel	Letztes Spiel
Ägypten	1	0	0	1	1:2	28.12.1958	28.12.1958
Albanien	14	13	1	0	38:10	08.04.1967	06.06.2001
Algerien	2	0	0	2			
Argentinien	20	6	5	9			
Armenien	2	2	0	0			
Aserbaidshan	4	4	0	0			
Australien	4	3	0	1			
Belgien	25	20	1	4			
Böhmen-Mähren	1	0	1	0			
Bolivien	1	1	0	0			
Bosnien-Herzegowina	2	1	1	0			


```

Ägypten 1 0 0 1 1:2 28.12.1958 28.12.1958
Albanien 14 13 1 0 38:10 08.04.1967 06.06.2001
Algerien 2 0 0 2 1:4 01.01.1964 16.06.1982
Argentinien 20 6 5 9 28:28 08.06.1958 15.08.2012
Armenien 2 2 0 0 9:1 09.10.1996 10.09.1997
Aserbaidshan 4 4 0 0 15:2 12.08.2009 07.06.2011
Australien 4 3 0 1 12:5 18.06.1974 29.03.2011
Belgien 25 20 1 4 58:26 16.05.1910 11.10.2011
Boehmen-Maehren 1 0 1 0 4:4 12.11.1939 12.11.1939
Bolivien 1 1 0 0 1:0 17.06.1994 17.06.1994
Bosnien-Herzegowina 2 1 1 0 4:2 11.10.2002 03.06.2010
Brasilien 21 4 5 12 24:39 05.05.1963 10.08.2011
Bulgarien 21 16 2 3 56:24 20.10.1935 21.08.2002
Chile 6 4 0 2 11:7 23.03.1960 20.06.1982
China 2 1 1 0 2:1 12.10.2005 29.05.2009
Costa-Rica 1 1 0 0 4:2 09.06.2006 09.06.2006
Daenemark 26 15 3 8 53:36 06.10.1912 17.06.2012
DDR 1 0 0 1 0:1 22.06.1974 22.06.1974
Ecuador 2 2 0 0 7:2 20.06.2006 29.05.2013
Elfenbeinküste 1 0 1 0 2:2 18.11.2009 18.11.2009
England 32 11 6 15 41:67 20.04.1908 27.06.2010
Estland 3 3 0 0 1:1 15.09.1935 29.03.1939
  
```

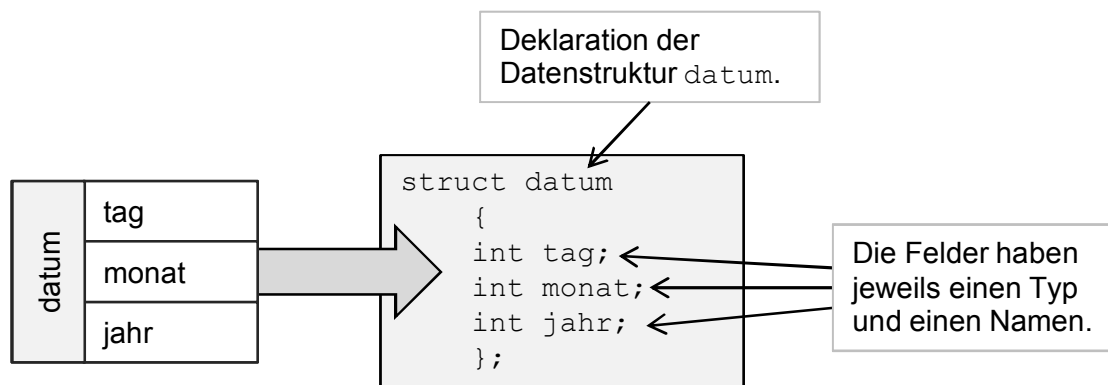
Laenderspiele.txt

Eine Zeile in dieser Datei bildet einen zusammengehörigen Datensatz, den man als Ganzes verarbeiten (zum Beispiel einlesen, ausgeben, ändern) will. Mit unseren bisherigen Mitteln ist das nicht möglich.

Deklaration von Datenstrukturen

In den beiden letzten Spalten der Länderspieltabelle stehen die Kalenderdaten für das erste und das letzte Spiel gegen die jeweils andere Nation. Wir wollen Tag, Monat und Jahr eines Datums so zusammenfassen, dass man ein Datum als Ganzes behandeln, aber auch auf gezielt auf Tag, Monat und Jahr zugreifen kann.

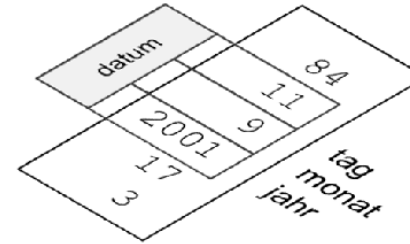
Dazu deklarieren wir eine Datenstruktur:



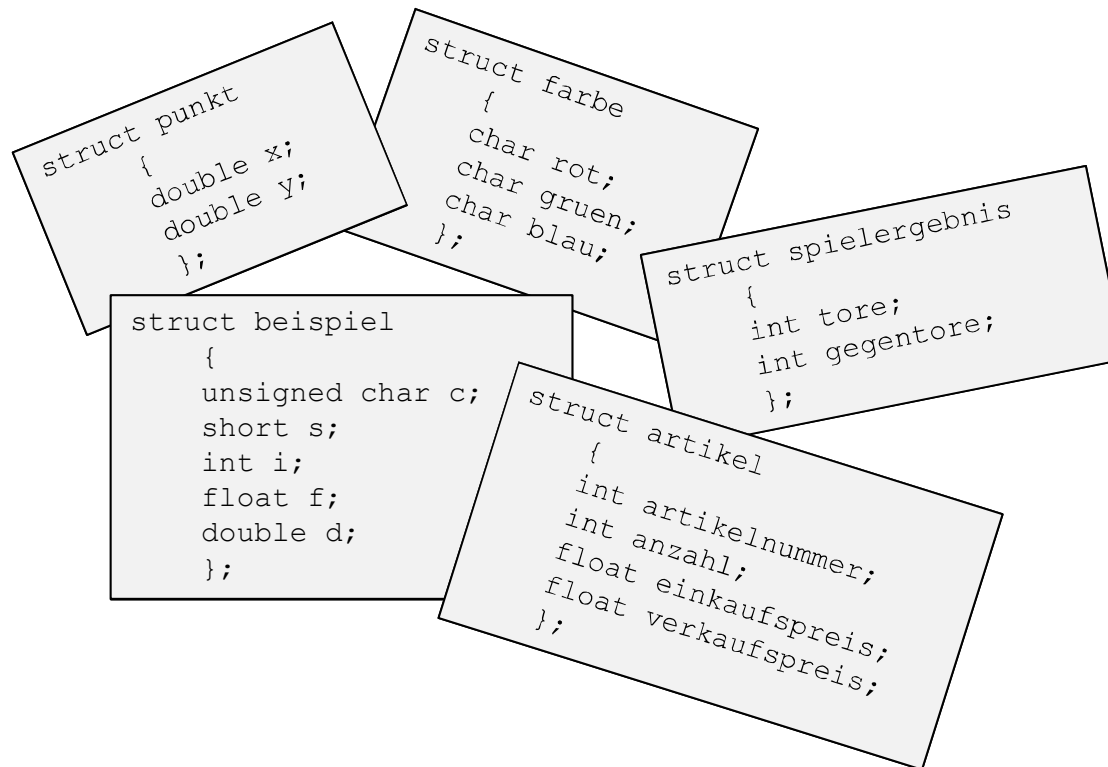
Erstes Spiel	Letztes Spiel
28.12.1958	28.12.1958
08.04.1967	06.06.2001
01.01.1964	16.06.1982
08.06.1958	15.08.2012
09.10.1996	10.09.1997
12.08.2009	07.06.2011
18.06.1974	29.03.2011
16.05.1910	11.10.2011
12.11.1939	12.11.1939
17.06.1994	17.06.1994
14.10.2002	03.06.2010

Datenstrukturen sind keine Daten

Mit der Deklaration einer Datenstruktur entstehen keine Daten und kein Code. Eine Datenstruktur ist nur Schablone, durch die wir auf unsere Daten blicken wollen. Die Schablone strukturiert die Daten.



Die elementaren Datentypen (char, int, float, double...) sind der Rohstoff, aus dem Datenstrukturen zusammengesetzt werden können.



Deklaration weiterer Datenstrukturen für die Länderspielbilanz

Bilanz							
Land	Spiele	gew.	unent.	verl.	Tore	Erstes Spiel	Letztes Spiel
Ägypten	1	0	0	1	1:2	28.12.1958	28.12.1958
Albanien	1					04.	
Algerien						01.	
Argentinien	2					08.06.	
Armenien							10.09.1997
Aserbaidshan	4						07.06.2011
Australien							09.03.2011
Belgien					58:26		
Böhmen-Mähren					4:4		
Bolivien					1:0		
Bosnien-Herzegowina	2	1	1	0	4:2	11.09.2002	03.06.2012

spiele

gesamt
gew
unent
verl

tore

dfb
gegner

datum

tag
monat
jahr

```

struct spiele
{
    int gesamt;
    int gew;
    int unent;
    int verl;
};
          
```

```

struct tore
{
    int dfb;
    int gegner;
};
          
```

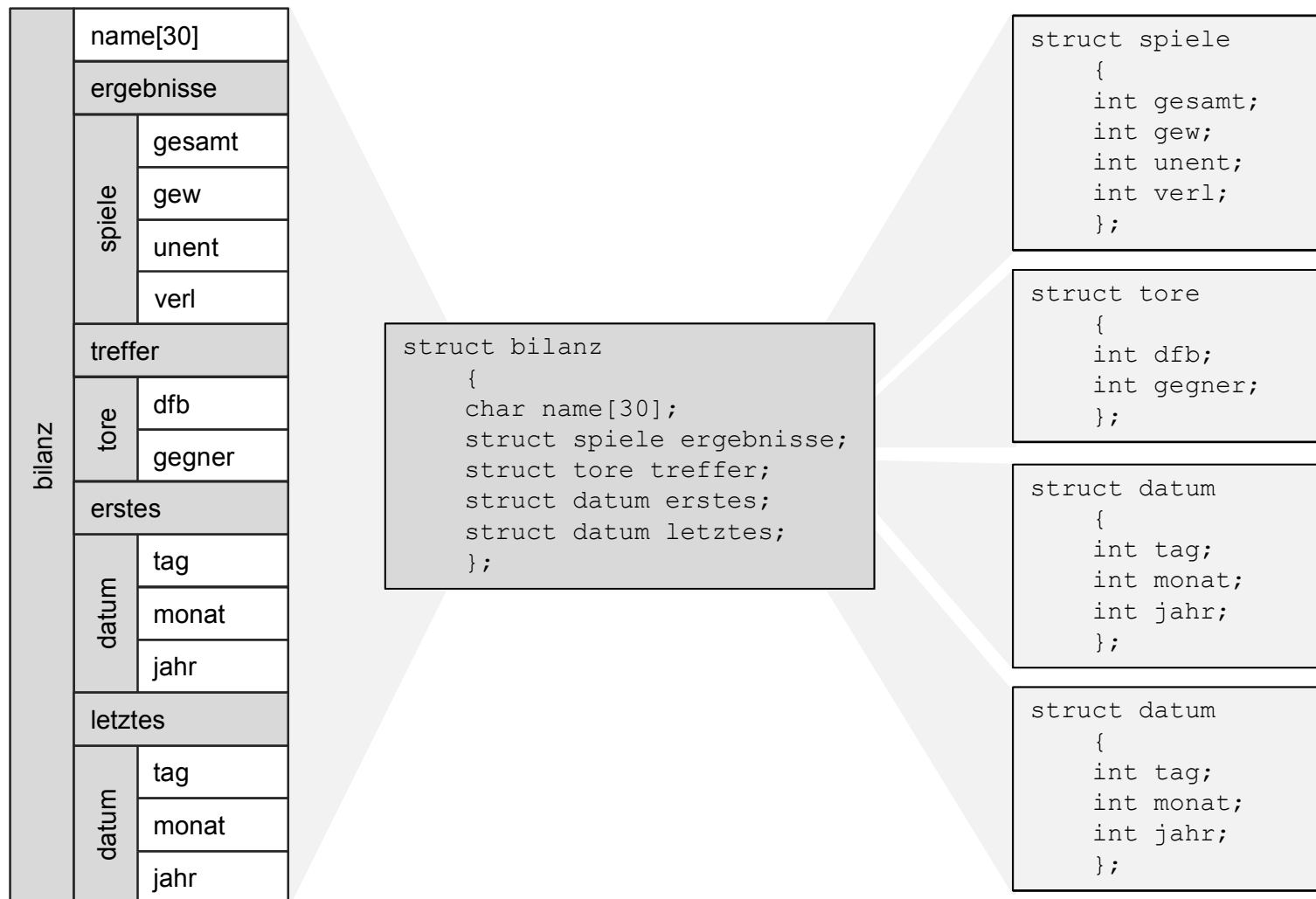
```

struct datum
{
    int tag;
    int monat;
    int jahr;
};
          
```

Nach Bedarf können aus allen Grunddatentypen Datenstrukturen zusammengestellt werden, auch wenn wir es in unserem Beispiel nur mit ganzen Zahlen zu tun haben.

Komplexere Strukturen

Datenstrukturen können Strukturen und Arrays enthalten. Zur Modellierung einer Zeile der Länderspieltabelle greifen wir auf die bereits deklarierten Teilstrukturen (`spiele`, `tore`, `datum`) zurück und fügen noch einen Array von 30 Zeichen für den Namen des Landes hinzu:



Bezeichner in Datenstrukturen

Alles in einer Datenstruktur hat einen Namen. Grundsätzlich gibt es zwei verschiedene Arten von Namen:

- Struktur-Namen (in der Grafik senkrecht geschrieben), wie `bilanz` oder `datum`. Mit diesen Namen werden neue Strukturen eindeutig benannt.
- Feld-Namen (in der Grafik waagerecht geschrieben), wie `monat` oder `treffer`. Diese Namen dienen zum Zugriff auf die Felder einer Datenstruktur

Die Struktur `bilanz` enthält zum Beispiel unter dem Namen `ergebnisse` eine Struktur `spiele`.

Die Struktur `datum` ist zweimal in der Struktur `bilanz` vorhanden. Auf das eine Datum kann unter dem Namen `erstes`, auf das zweite unter dem Namen `letztes` zugegriffen werden.

Wie ein Zugriff auf die Daten konkret aussieht, werden wir später sehen. Noch gibt es ja gar keine Daten sondern nur Schablonen mit Struktur- und Zugriffsinformationen.

bilanz	name[30]	
	ergebnisse	
	spiele	gesamt
		gew
		unent
		verl
	treffer	
	tore	dfb
		gegner
	erstes	
	datum	tag
		monat
		jahr
	letztes	
	datum	tag
		monat
		jahr

Gesamtmodell

Um die Tabelle mit den Länderspielbilanzen als Ganzes zu modellieren, werden wir jetzt noch einen Array von ausreichend vielen (100) Bilanzen erstellen und zusätzlich speichern, wie viele Einträge in diesem Array gültig sind.

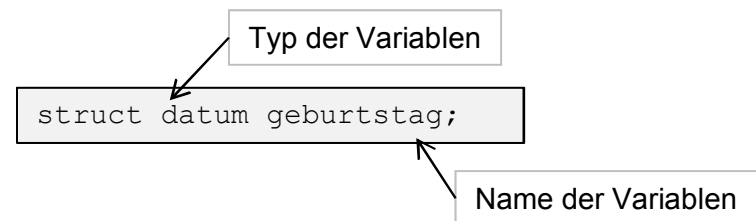


Beachten Sie, dass die Arrays in diesem Beispiel auf die zu erwartende Maximallast (maximal 29 Buchstaben im Ländernamen, maximal 100 verschiedene Länder) ausgelegt sind. Das ist eine Beschränkung, von der wir uns später befreien werden.

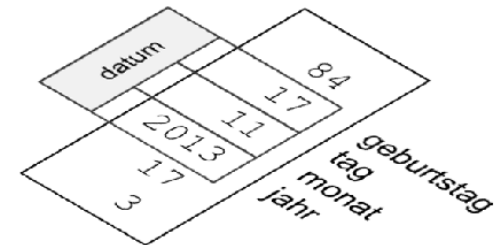
Variablendefinition

Durch die Deklaration einer Datenstruktur wird nur ein neuer Datentyp eingeführt. Üblicherweise findet man Datenstruktur-Deklarationen in Headerdateien, die dann von allen Quelldateien, die diese Datenstrukturen verwenden wollen, inkludiert werden. Werden Datenstruktur-Deklarationen nur in einer einzigen Quelldatei benötigt, können sie auch dort, typischerweise am Anfang der Datei, stehen.

Konkrete Daten einer bestimmten Struktur erhält man erst, wenn man eine Variable definiert:



Jetzt ist ein konkretes Datum (`geburtstag`) entstanden, das auch schon bei der Definition mit Werten gefüllt werden kann:



Definition und Initialisierung komplexer Strukturvariablen

Auch komplexe, verschachtelte Strukturen können angelegt und initialisiert werden.
Man folgt einfach der durch die Schablone vorgegebenen Struktur.

```
struct bilanz beispiel =  
{ "Lummerland",  
  { 6, 1, 2, 3},  
  { 13, 17},  
  { 2, 3, 1975},  
  { 24, 12, 2000}  
};
```

```
struct bilanz  
{  
    char name[30];  
    struct spiele ergebnisse;  
    struct tore treffer;  
    struct datum erstes;  
    struct datum letztes;  
};
```

```
struct spiele  
{  
    int gesamt;  
    int gew;  
    int unent;  
    int verl;  
};
```

```
struct tore  
{  
    int dfb;  
    int gegner;  
};
```

```
struct datum  
{  
    int tag;  
    int monat;  
    int jahr;  
};
```

```
struct datum  
{  
    int tag;  
    int monat;  
    int jahr;  
};
```

Zuweisung von Datenstrukturen

Die Werte einer Variablen können einer anderen Variablen zugewiesen werden, egal, ob die Variablen nur auf einem einfachen Datentyp oder einer komplexen Struktur basieren. Wichtig ist, dass bei einer Zuweisung auf der linken und rechten Seite des Gleichheitszeichens der gleiche Datentyp steht.

```
struct datum heute = {31, 8, 2014};  
struct datum morgen;  
  
morgen = heute; ←  
  
printf( "%d.%d.%d\n", morgen.tag, morgen.monat, morgen.jahr);
```

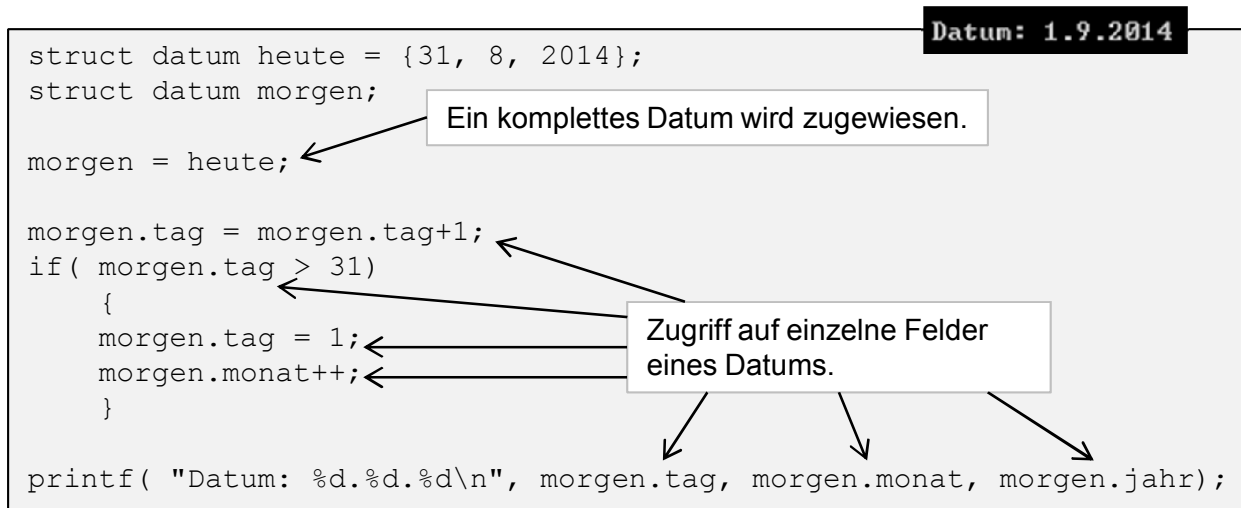
Ein komplettes Datum wird zugewiesen.

31.8.2014

Operationen wie zum Beispiel Größenvergleich (<, >) oder arithmetische Operationen kann man auf Datenstrukturen nicht ausführen (da müssen wir uns noch bis zur objektorientierten Programmierung gedulden). Wie sollte der Compiler auch wissen, wie etwa der Vergleich zweier Kalenderdaten, im Sinne eines Vorher-Nachher-Vergleichs, durchgeführt werden sollte.

Direktzugriff auf die Felder einer Datenstruktur

Zum direkten Zugriff auf die Felder einer Datenstruktur dient der Punkt-Operator (.):



```
struct datum heute = {31, 8, 2014};
struct datum morgen;

morgen = heute;

morgen.tag = morgen.tag+1;
if( morgen.tag > 31)
{
    morgen.tag = 1;
    morgen.monat++;
}

printf( "Datum: %d.%d.%d\n", morgen.tag, morgen.monat, morgen.jahr);
```

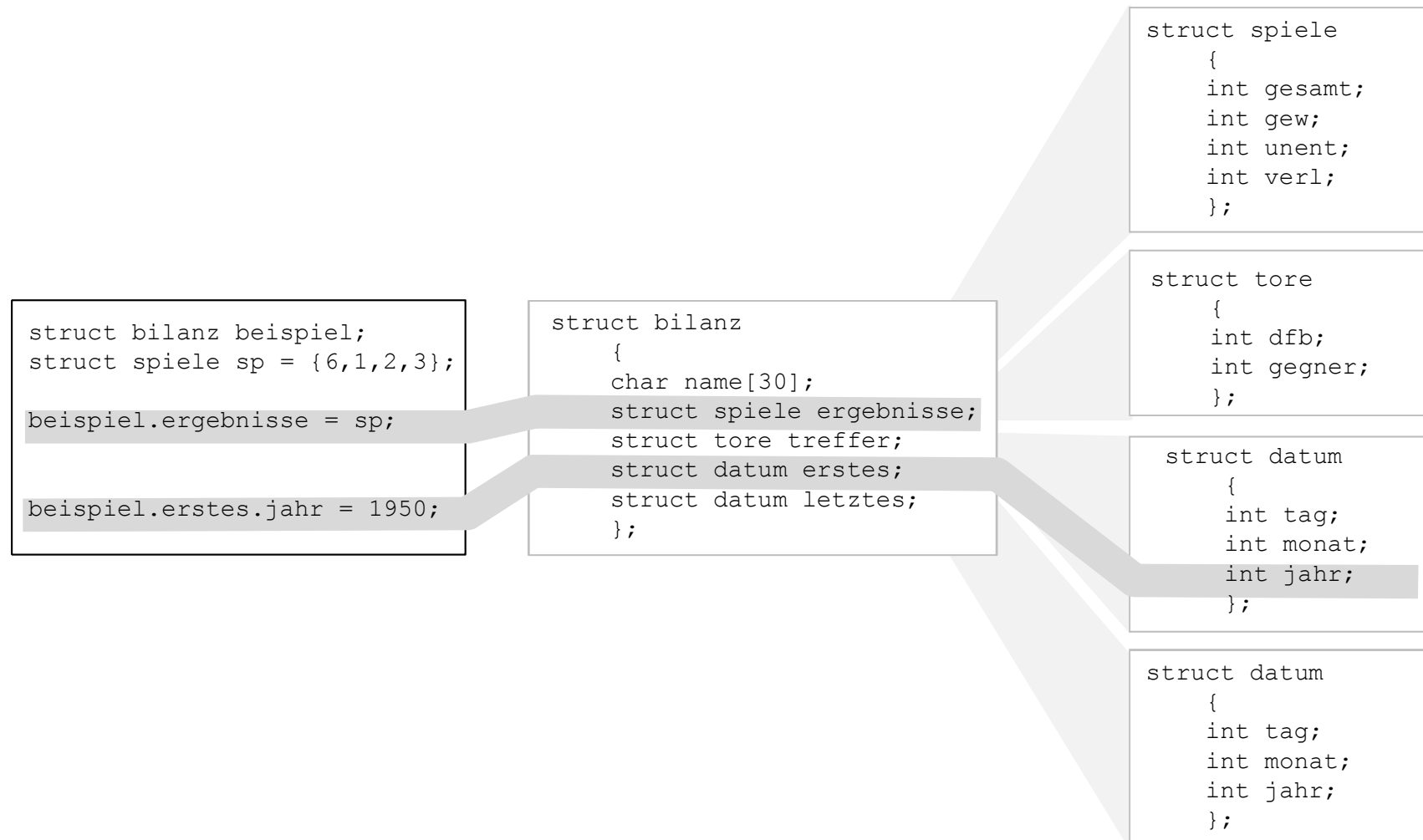
Datum: 1.9.2014

Ein komplettes Datum wird zugewiesen.

Zugriff auf einzelne Felder eines Datums.

Zugriff in verschachtelte Datenstrukturen

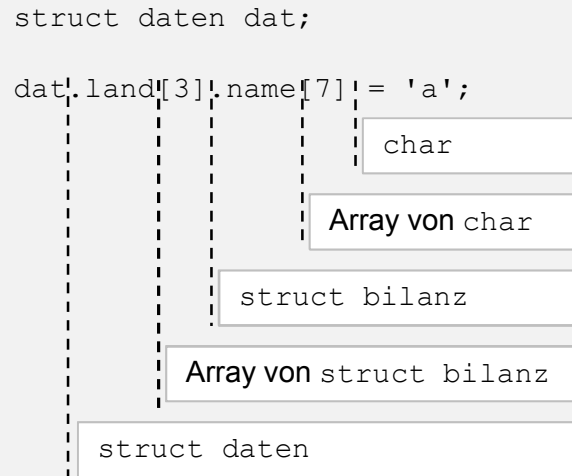
Schritt für Schritt kann man mit dem Punkt-Operator in eine Datenstruktur hineinzoomen, bis man auf dem Level angekommen ist, auf dem man arbeiten möchte, egal, wie tief die Strukturen verschachtelt sind



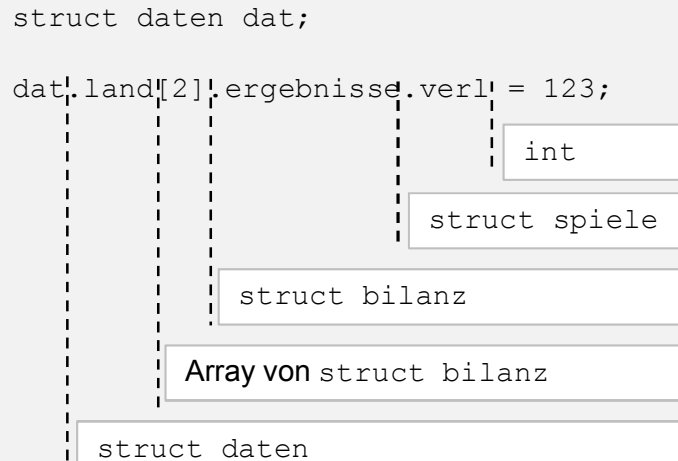
Datentypen beim Zugriff

Wichtig ist, immer im Blick zu behalten, welchen Datentyp man auf welcher Zugriffsstufe jeweils erhält, damit man weiß, welche Operationen man auf dem jeweiligen Level ausführen kann.

```
struct daten dat;  
dat.land[3].name[7] = 'a';
```



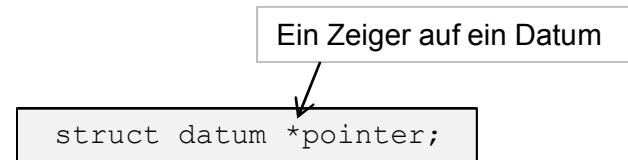
```
struct daten dat;  
dat.land[2].ergebnisse.verl = 123;
```



```
struct datum  
{  
    int tag;  
    int monat;  
    int jahr;  
};  
  
struct spiele  
{  
    int gesamt;  
    int gew;  
    int unent;  
    int verl;  
};  
  
struct tore  
{  
    int dfb;  
    int gegner;  
};  
  
struct bilanz  
{  
    char name[30];  
    struct spiele ergebnisse;  
    struct tore treffer;  
    struct datum erstes;  
    struct datum letztes;  
};  
  
struct daten  
{  
    int anzahl;  
    struct bilanz land[100];  
};
```

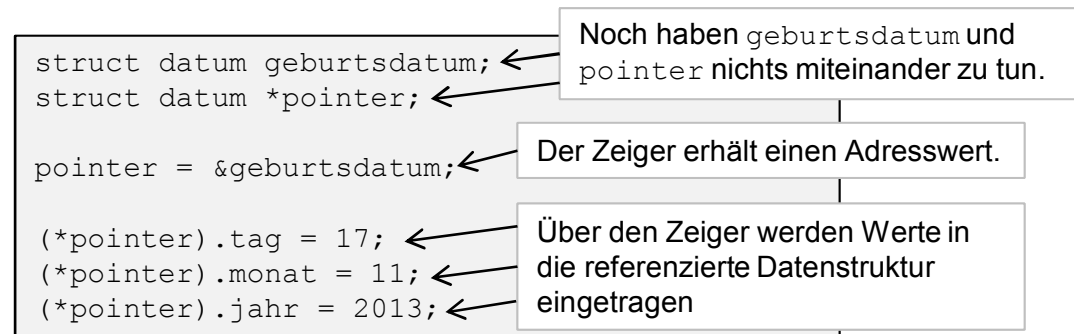
Indirektzugriff auf Datenstrukturen

Wir können auch Zeiger auf Datenstrukturen anlegen, so wie wir bereits Zeiger auf die Grunddatentypen angelegt hatten:



Bei `pointer` handelt es sich nicht um eine Datenstruktur mit Feldern `tag`, `monat` und `jahr`, sondern `pointer` ist ein Zeiger, der die Adresse einer solchen Datenstruktur enthält.

Der Zeiger ist unbrauchbar, solange ihm nicht die Adresse einer konkreten Datenstruktur zugewiesen wird:



Mit dem Adressoperator (`&`) kann man die Adresse einer Variablen ermitteln und mit dem Dereferenzierungsoperator (`*`) kann man über eine Adresse auf eine Variable zugreifen.

Der Points-Operator

Ist p ein Zeiger auf eine Datenstruktur und x ein Feld dieser Datenstruktur, so lässt sich auf das Feld mit den beiden gleichwertigen Ausdrücken

$(*p) . x$ bzw. $p \rightarrow x$

zugreifen. Beide Ausdrücke sind dabei als R-Value und L-Value – also sowohl auf der rechten als auch auf der linken Seite einer Zuweisung – geeignet.

Den Ausdruck $p \rightarrow x$ lesen wir als » p points x «

Damit kann man den Strukturzugriff eleganter formulieren:

Zugriff mit *-Operator

```
struct datum geburtsdatum;  
struct datum *pointer;  
  
pointer = &geburtsdatum;  
  
(*pointer).tag = 17;  
(*pointer).monat = 11;  
(*pointer).jahr = 2013;
```

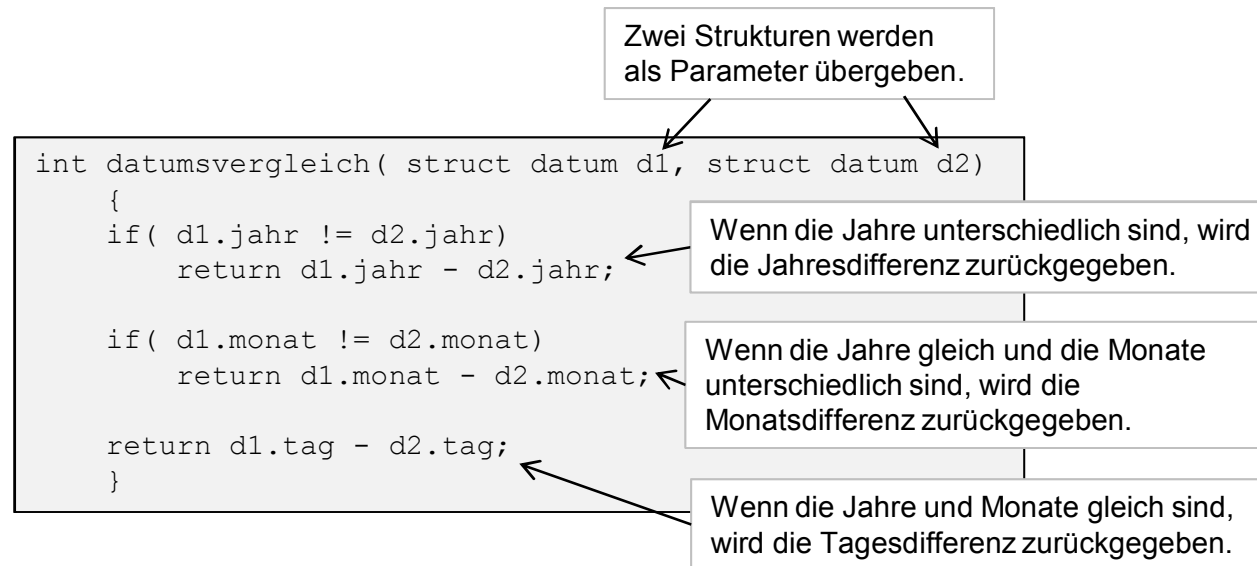
Zugriff mit Points-Operator

```
struct datum geburtsdatum;  
struct datum *pointer;  
  
pointer = &geburtsdatum;  
  
pointer->tag = 17;  
pointer->monat = 11;  
pointer->jahr = 2013;
```

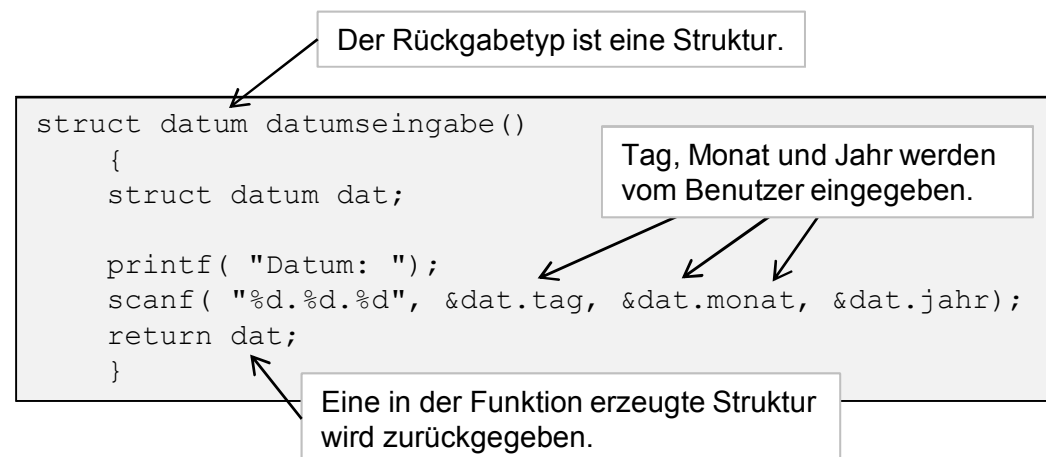
Den wahren Wert von Zeigern erkennt man aber erst im Zusammenhang mit Funktionen und dynamischen Datenstrukturen.

Datenstrukturen und Funktionen 1

Datenstrukturen können als Parameter an Funktionen übergeben werden:

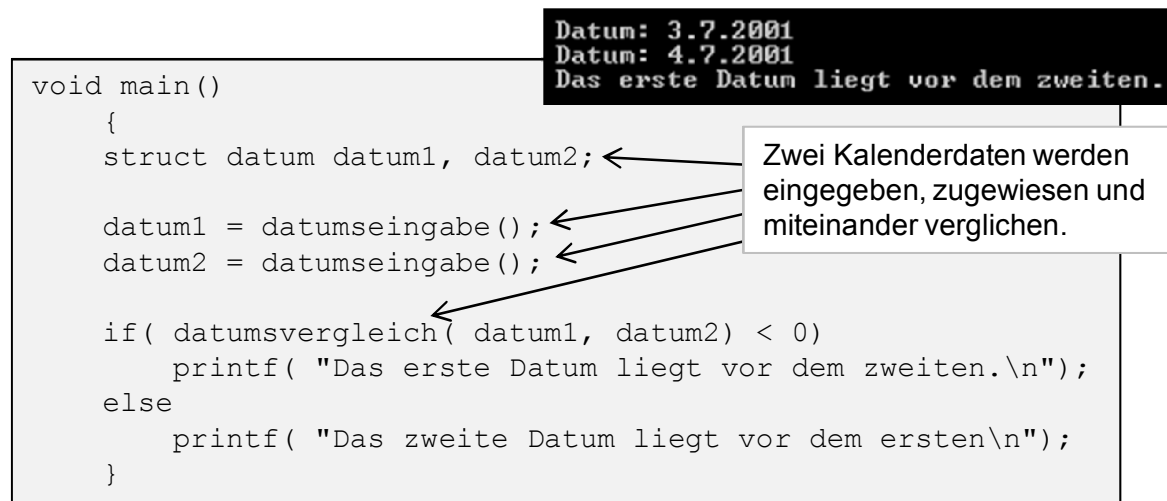


Datenstrukturen können als Returnwert von Funktionen zurückgegeben werden:



Datenstrukturen und Funktionen 2

Die Funktionen `datumseingabe` und `datumsvergleich` kann man zum Beispiel wie folgt verwenden:



Beim Funktionsaufruf werden Kopien der übergebenen Daten auf dem Stack erzeugt, die Funktion arbeitet mit diesen Kopien und beim Rücksprung werden die Daten auf dem Stack wieder beseitigt.

Datenstrukturen können sehr groß sein, sodass bei einem Funktionsaufruf gegebenenfalls große Datenmengen, zumeist überflüssigerweise, dupliziert werden.

Durch die Verwendung von Zeigern kann man dies vermeiden.

Zeiger als Funktionsparameter

Umstellung der Funktion `datumsvergleich` auf die konsequente Verwendung von Zeigern:

Zwei Zeiger auf Strukturen werden
als Parameter übergeben.

```
int datumsvergleich( struct datum *pd1, struct datum *pd2)
{
    if( pd1->jahr != pd2->jahr)
        return pd1->jahr - pd2->jahr;
    if( pd1->monat != pd2->monat)
        return pd1->monat - pd2->monat;
    return pd1->tag - pd2->tag;
}
```

Der Zugriff auf die Daten erfolgt mit
dem Pfeil-Operator, ansonsten hat
sich nichts geändert.

An der Schnittstelle werden jetzt nur noch zwei Zeiger übergeben.

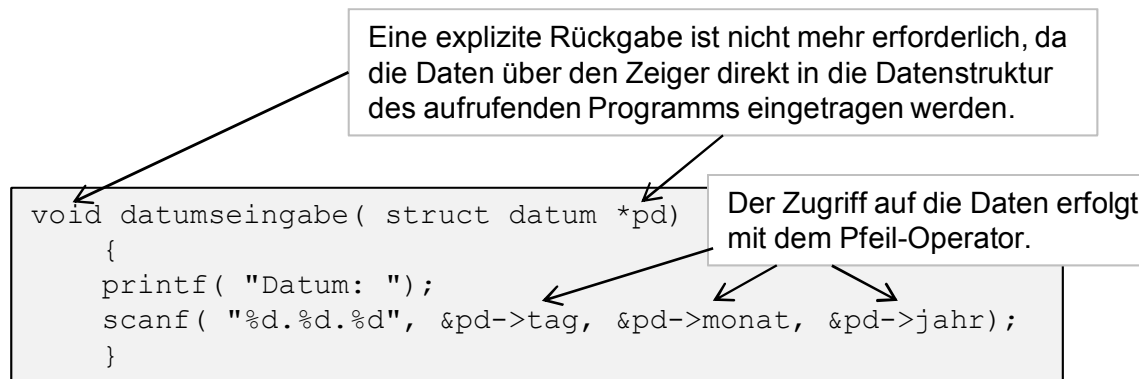
Die Zeiger werden auch als Kopien übergeben, aber ein Zeiger ist in der Regel sehr viel kleiner als eine Datenstruktur.

Achtung: Die Funktion kann jetzt die Daten im rufenden Programm über die Zeiger verändern.

Zeiger als Rückgabewert von Funktionen

Eine Funktion kann auch einen Zeiger als Returnwert zurückgeben. Aber, eine Funktion darf keinen Zeiger auf eine lokale Datenstruktur zurückgeben, da eine solche Datenstruktur beim Rücksprung aus der Funktion beseitigt wird. Der Zeiger würde dann "ins Leere" zeigen.

Statt dessen kann das rufende Programm einen Zeiger auf eine Datenstruktur übergeben, die dann vom aufgerufenen Programm mit Werten gefüllt wird:



Verwendung von Funktionen mit Zeigerparametern

Das Hauptprogramm mit konsequenter Verwendung von Zeigern:

```
void main()
{
    struct datum datum1, datum2;

    datumseingabe( &datum1);
    datumseingabe( &datum2);

    if( datumsvergleich( &datum1, &datum2) < 0)
        printf( "Das erste Datum liegt vor dem zweiten.\n");
    else
        printf( "Das zweite Datum liegt vor dem ersten\n");
}
```

```
Datum: 3.7.2001
Datum: 4.7.2001
Das erste Datum liegt vor dem zweiten.
```

Statt die Daten zu kopieren,
werden jetzt Adressen
übergeben.

Alle Daten liegen jetzt im Hauptprogramm. Die Unterprogramme arbeiten nur mit Zeigern auf den Daten im Hauptprogramm. An der Schnittstelle werden nur noch Zeiger übertragen.

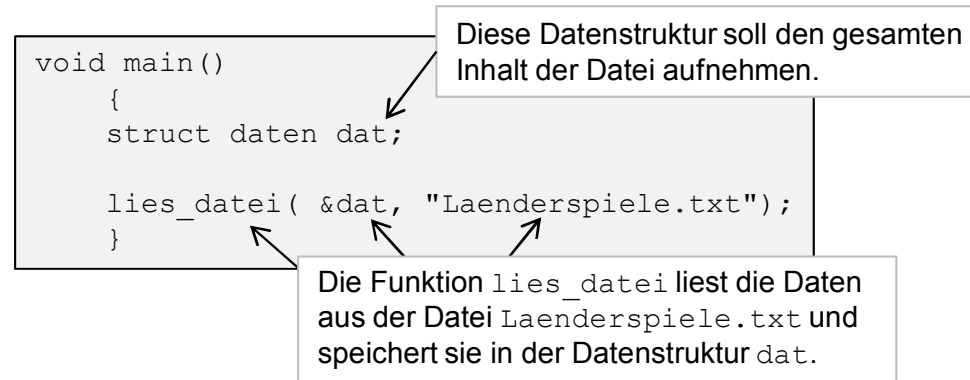
Einlesen von Daten aus einer Datei in eine Datenstruktur 1

Wir haben eine Datenstruktur bereitgestellt, in die wir den kompletten Inhalt der Länderspieltabelle einlesen können:

Bilanz							
Land	Spiele	gew.	unent.	verl.	Tore	Erstes Spiel	Letztes Spiel
Ägypten	1	0	0	1	1:2	28.12.1958	28.12.1958
Albanien			1			4.1.1967	06.06.2001
Algerien			0			1.1.1964	16.06.1982
Argentinien			5			6.1.1958	15.08.2012
Armenien							
Aserbaidschan					15:2	12	
Australien							
Belgien						10	11.10.2011
Böhmen-Mähren	1	0	1			39	12.11.1939
Bolivien	1	1	0	0	1:0	17.06.1994	17.06.1994
Bosnien-Herzegowina	2	1	1	0	1:2	14.10.2002	03.06.2010

Die Daten habe ich dazu in einer einfachen Textdatei (Laenderspiele.txt) bereitgestellt. Ich habe dabei Umlaute und Trennzeichen aus den Ländernamen entfernt.

Einlesen von Daten aus einer Datei in eine Datenstruktur 2

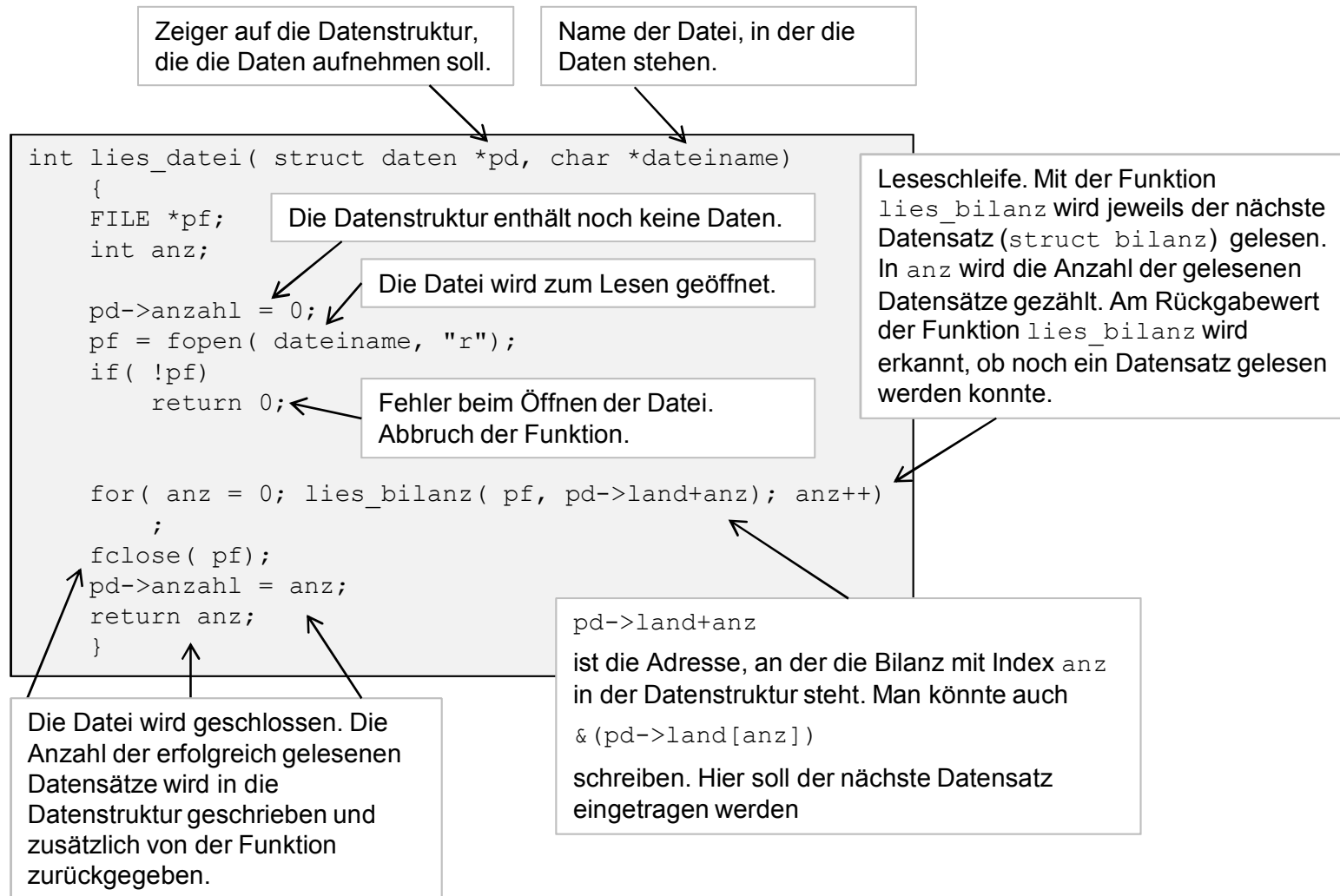


Erinnerung an einige wichtige Dateioperationen:

- `fopen` Öffnen einer Datei
- `feof` Test, ob hinter dem Dateiende gelesen wurde
- `fscanf` Einlesen von Daten aus der Datei ähnlich `scanf`
- `fclose` Schließen der Datei

Wichtig ist der Dateihandle (Datentyp `FILE`), den wir beim Öffnen der Datei als Returnwert von `fopen` erhalten und bei allen Zugriffsfunktionen auf die Datei als Parameter verwenden müssen.

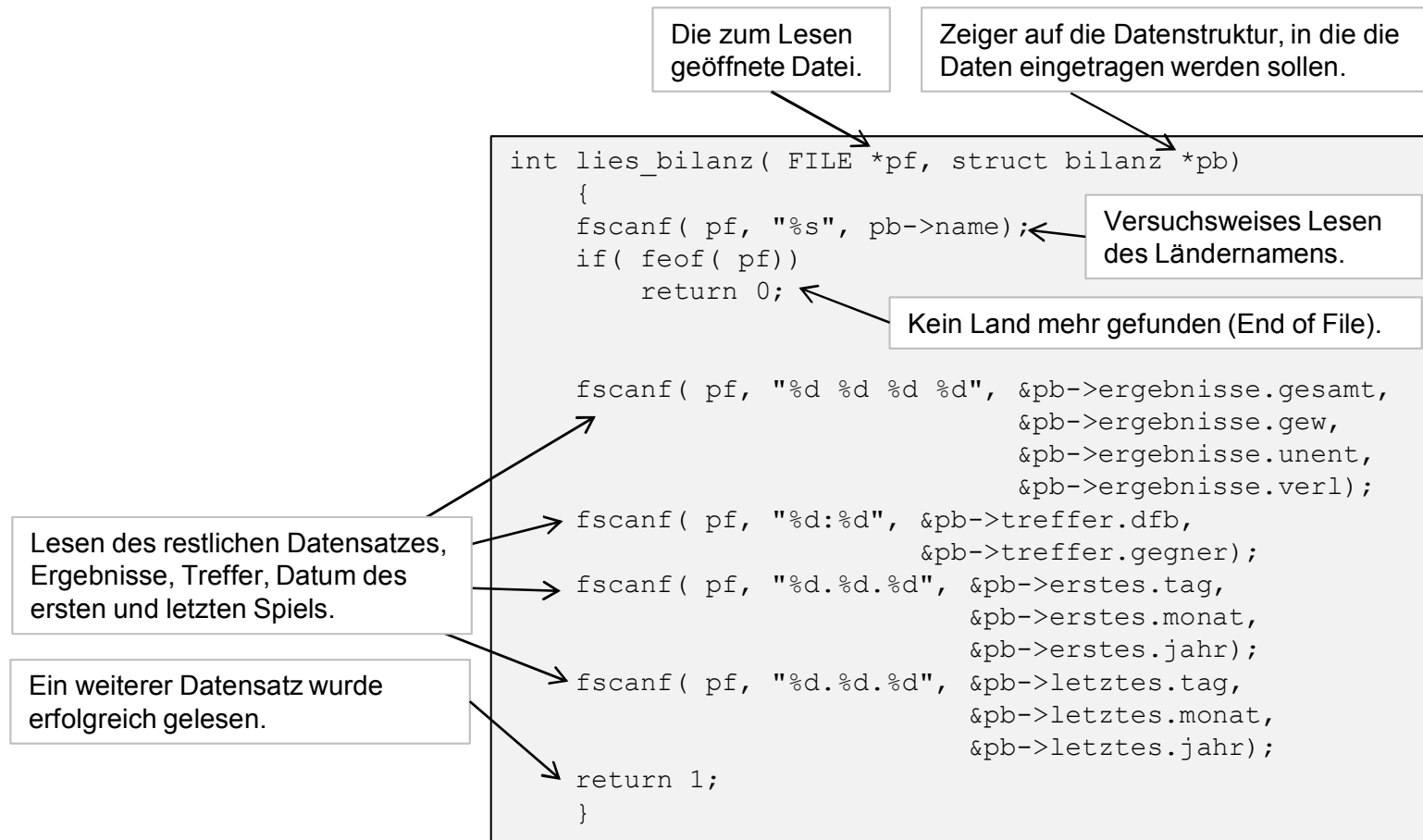
Einlesen von Daten aus einer Datei in eine Datenstruktur 3



Es fehlt noch die Implementierung der Funktion `lies_bilanz`, mit der eine Bilanz (= Zeile in der Datei) in die Struktur `bilanz` eingelesen wird.

Einlesen von Daten aus einer Datei in eine Datenstruktur 4

Die Funktion `lies_bilanz` erhält einen Zeiger auf eine Struktur `bilanz` und füllt die Struktur mit Daten aus der Datei:




Ob ein Datensatz erfolgreich gelesen werden konnte, erkennt das rufende Programm am Returnwert.

Ausgabe gesamten Datenstruktur auf dem Bildschirm

Zur Ausgabe iterieren wir über die gesamte Datenstruktur und geben jeden Datensatz mit der Funktion `print_bilanz` aus:

```
void print_daten( struct daten *pd)
{
    int i;

    for( i = 0; i < pd->anzahl; i++)
        print_bilanz( pd->land + i);
}
```



```
void print_bilanz( struct bilanz *pb)
{
    printf( "%-25s", pb->name);
    printf( " %3d %3d %3d %3d", pb->ergebnisse.gesamt,
                                   pb->ergebnisse.gew,
                                   pb->ergebnisse.unent,
                                   pb->ergebnisse.verl);

    printf( " %4d:%-4d", pb->treffer.dfb,
                                   pb->treffer.gegner);
    printf( " %02d.%02d.%4d", pb->erstes.tag,
                                   pb->erstes.monat,
                                   pb->erstes.jahr);
    printf( " %02d.%02d.%4d", pb->letztes.tag,
                                   pb->letztes.monat,
                                   pb->letztes.jahr);

    printf( "\n");
}
```

Das Ergebnis

Der Inhalt der Datei ist jetzt vollständig in die interne Datenstruktur übertragen.

Durch die Datenstruktur sind die Daten so aufbereitet, dass im Programm flexibel auf beliebige Teile der Daten lesend und schreibend zugegriffen werden kann.

Geänderte Daten können in die Datei zurückgeschrieben werden.

```
Aegypten 1 0 0 1 1:2 28.12.1958 28.12.1958
Albanien 14 13 1 0 38:10 08.04.1967 06.06.2001
Algerien 2 0 0 2 1:4 01.01.1964 16.06.1982
Argentinien 20 6 5 9 28:28 08.06.1958 15.08.2012
Armenien 2 2 0 0 9:1 09.10.1996 10.09.1997
Aserbajdschan 4 4 0 0 15:2 12.08.2009 07.06.2011
Australien 4 3 0 1 12:5 18.06.1974 29.03.2011
Belgien 25 20 1 4 58:26 16.05.1910 11.10.2011
Bohmen-Maehren 1 0 1 0 4:4 12.11.1939 12.11.1939
Bolivien 1 1 0 0 1:0 17.06.1994 17.06.1994
Bosnien-Herzegowina 2 1 1 0 4:2 11.10.2002 03.06.2010
Brasilien 21 4 5 12 24:39 05.05.1963 10.08.2011
Bulgarien 21 16 2 3 56:24 20.10.1935 21.08.2002
Chile 6 4 0 2 11:7 23.03.1960 20.06.1982
China 2 1 1 0 2:1 12.10.2005 29.05.2009
Costa-Rica 1 1 0 0 4:2 09.06.2006 09.06.2006
Daenemark 26 15 3 8 53:36 06.10.1912 17.06.2012
DDR 1 0 0 1 0:1 22.06.1974 22.06.1974
Ecuador 2 2 0 0 7:2 20.06.2006 29.05.2013
Elfenbeinkueste 1 0 1 0 2:2 18.11.2009 18.11.2009
England 32 11 6 15 41:67 20.04.1908 27.06.2010
Estland 3 3 0 0 1:1 15.09.1935 29.03.1939
```

```
void main()
{
    struct daten dat;

    lies_datei( &dat, "Laenderspiele.txt");
    print_daten( &dat);
}
```

```
Aegypten      1  0  0  1  1:2  28.12.1958 28.12.1958
Albanien     14 13  1  0 38:10  08.04.1967 06.06.2001
Algerien      2  0  0  2  1:4  01.01.1964 16.06.1982
Argentinien  20  6  5  9 28:28  08.06.1958 15.08.2012
Armenien      2  2  0  0  9:1   09.10.1996 10.09.1997
Aserbajdschan 4  4  0  0 15:2  12.08.2009 07.06.2011
Australien    4  3  0  1 12:5  18.06.1974 29.03.2011
Belgien     25 20  1  4 58:26  16.05.1910 11.10.2011
Bohmen-Maehren 1  0  1  0  4:4  12.11.1939 12.11.1939
Bolivien      1  1  0  0  1:0  17.06.1994 17.06.1994
Bosnien-Herzegowina 2 1 1 0 4:2  11.10.2002 03.06.2010
```

Verwendung der Datenstruktur

Wer ist der Lieblingsgegner der deutschen Mannschaft, d.h. die Mannschaft, gegen die bisher am häufigsten gespielt wurde?

```
int Lieblingsgegner( struct daten *pd)
{
    int i;
    int index;
    int max = -1;

    for( i = 0; i < pd->anzahl; i++)
    {
        if( pd->land[i].ergebnisse.gesamt > max)
        {
            max = pd->land[i].ergebnisse.gesamt;
            index = i;
        }
    }
    return index;
}
```

Index des gesuchten Lieblingsgegners.

Schleife über alle Länderbilanzen.

Wenn ein Gegner mit mehr Spielen gefunden wird, dann speichere das neue Maximum und den Index des Gegners.

Gib den Index des Lieblingsgegners zurück.

Beachten Sie

`dat.land+i`

ist das gleiche wie

`&(dat.land[i])`

also die Adresse der Bilanz des Landes mit dem Index `i`.

```
void main()
{
    struct daten dat;
    int i;

    lies_datei( &dat, "Laenderspiele.txt");

    i = Lieblingsgegner( &dat);
    printf( "\nLieblingsgegner\n");
    print_bilanz( dat.land+i);
}
```

Berechne den Index des Lieblingsgegners.

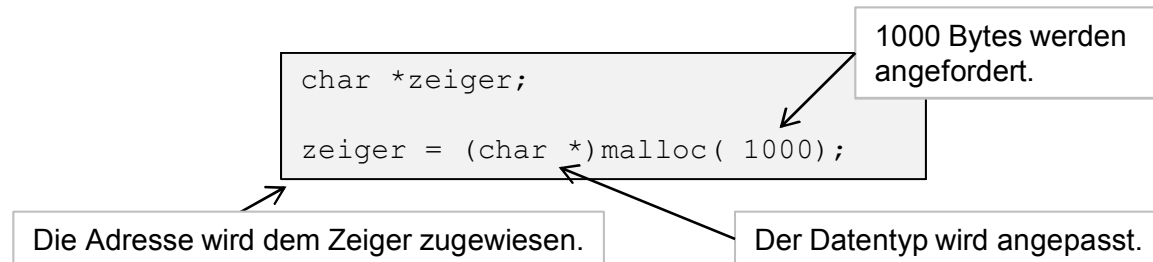
Gib die Bilanz des Lieblingsgegners aus

Lieblingsgegner
Schweiz

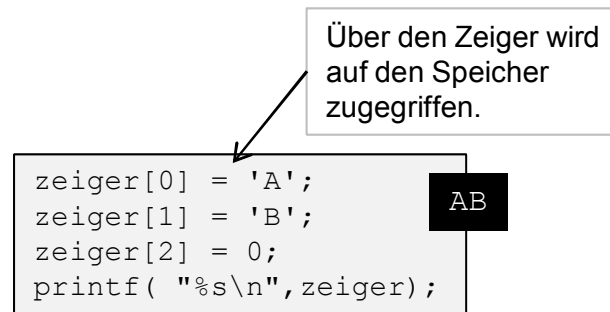
51 36 6 9 138:65 05.04.1908 26.05.2012

Allokieren von Speicher zur Laufzeit

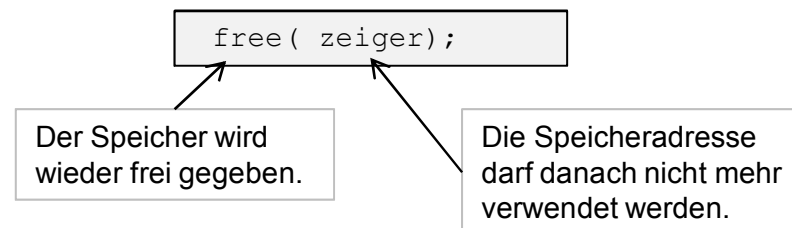
Wenn wir in einem Programm zur Laufzeit 1000 Bytes benötigen, um dort etwa einen Text zu speichern, können wir über die Funktion `malloc` den dazu erforderlichen Speicher anfordern.



Als Ergebnis des Funktionsaufrufes erhalten wir die Adresse des für uns reservierten Speichers. Diese Adresse müssen wir in einem Zeiger speichern, damit wir über diesen Zeiger auf den Speicher zugreifen können:

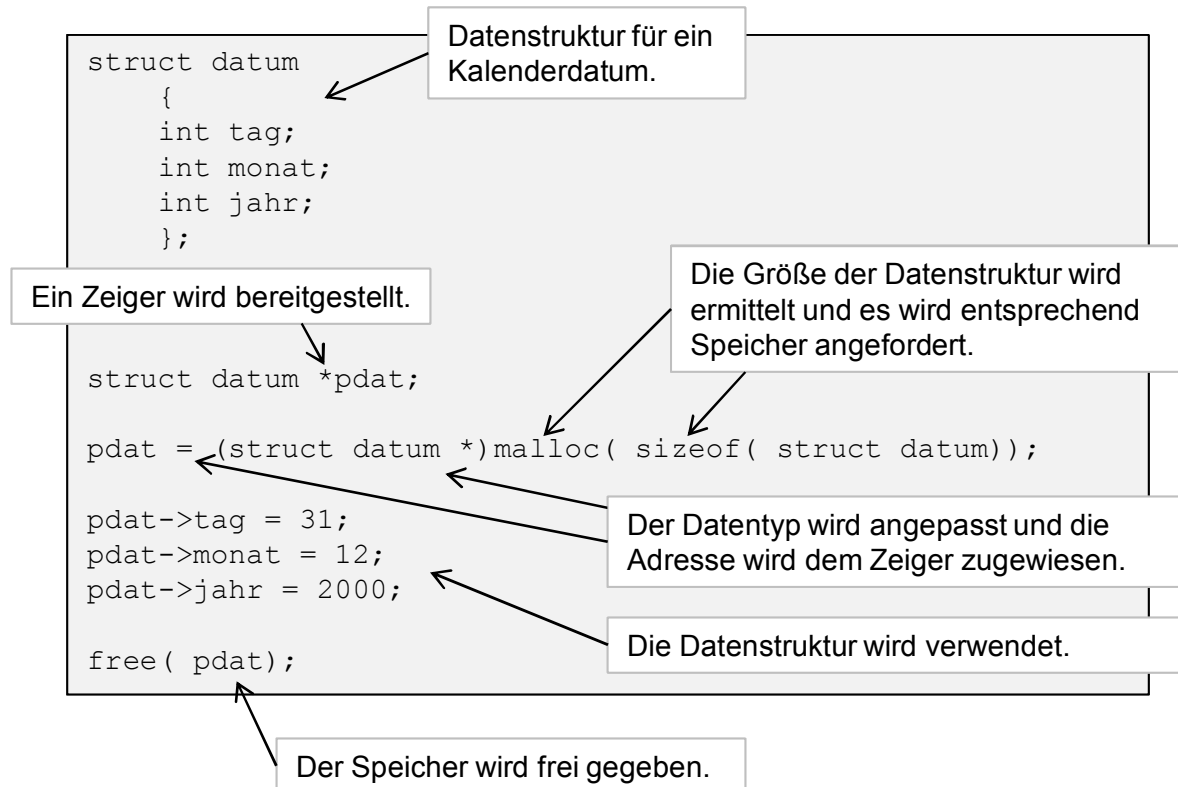


Wird der Speicher nicht mehr benötigt, wird er wieder an das Laufzeitsystem zurückgegeben:



Dynamische Datenstrukturen

Auch der Speicher für Datenstrukturen kann zur Laufzeit (dynamisch) allokiert werden:



In der Regel erfolgt die Freigabe des Speichers natürlich nicht unmittelbar nach einer einmaligen Verwendung sondern dann, wenn die Datenstruktur nicht mehr benötigt wird. Das kann an einer ganz anderen Stelle im Programm, unter Umständen auch ganz am Ende des Programms sein.

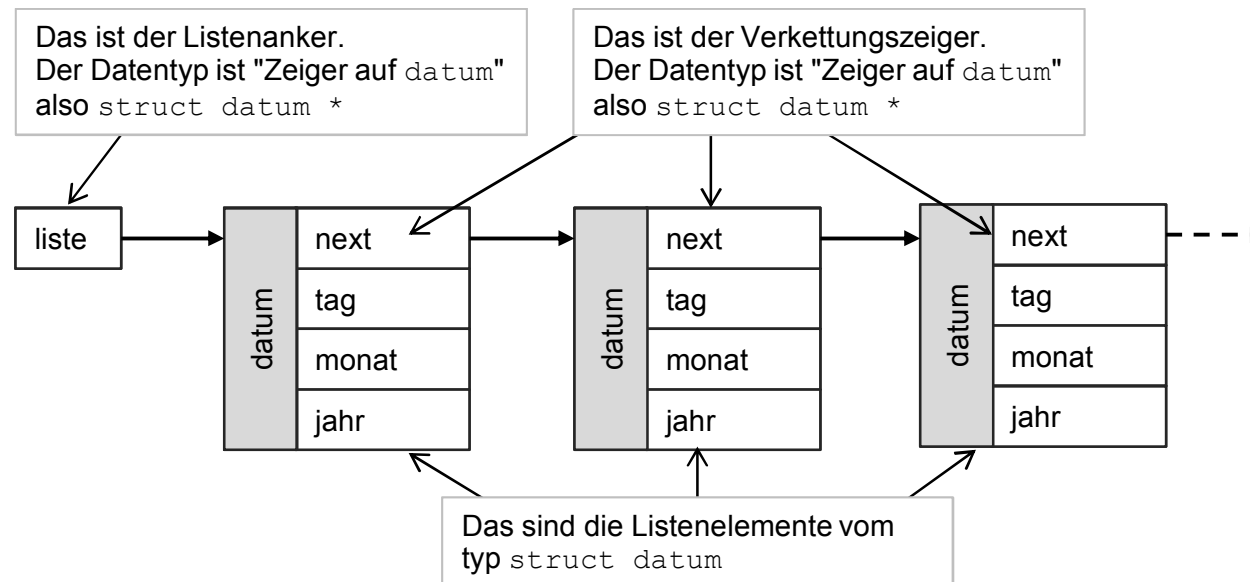
Was nützt das dynamische Allokieren von Datenstrukturen, wenn man für jede allokierte Struktur einen zur Compilezeit definierten Zeiger benötigt?

Listen

Was nützt das dynamische Allokieren von Datenstrukturen, wenn man für jede allokierte Struktur einen zur Compilezeit definierten Zeiger benötigt?

Antwort: Man legt den Zeiger mit in die Datenstruktur → Liste

Mit einer Liste kann man eine prinzipiell unbegrenzte Anzahl Datenstrukturen verwalten:



Deklaration einer geeigneten Datenstruktur:

Eine Liste wird durch den Zeigerwert 0
(oder auch symbolische Konstante NULL)
abgeschlossen.

```
struct datum
{
    struct datum *next;
    int tag;
    int monat;
    int jahr;
};
```

Zeiger auf das nächste
Kalenderdatum.

Aufbau einer Liste

In der folgenden Funktion kann der Benutzer beliebig viele Kalenderdaten eingeben. Die Funktion baut eine Liste mit den Daten auf:

```
struct datum *eingabe()
{
    struct datum *anker = NULL;
    struct datum *pneu;
    int weiter;

    for( ; ; )
    {
        printf( "Noch ein Datum? ");
        scanf( "%d", &weiter);
        if( !weiter)
            break;
        pneu= (struct datum *)malloc( sizeof( struct datum));
        printf( "Datum: ");
        scanf( "%d.%d.%d", &pneu->tag, &pneu->monat, &pneu->jahr);

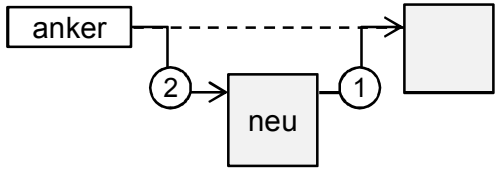
        pneu->next = anker;
        anker = pneu;
    }
    return anker;
}
```

Dies ist der Listenanker. Die Liste ist noch leer

Hilfszeiger für neue Listenelemente.

Der Benutzer will noch ein weiteres Datum eingeben, also wird eine neue Datenstruktur allokiert und mit Werten gefüllt.

Das neue Element wird vorn in die Liste eingekettet:

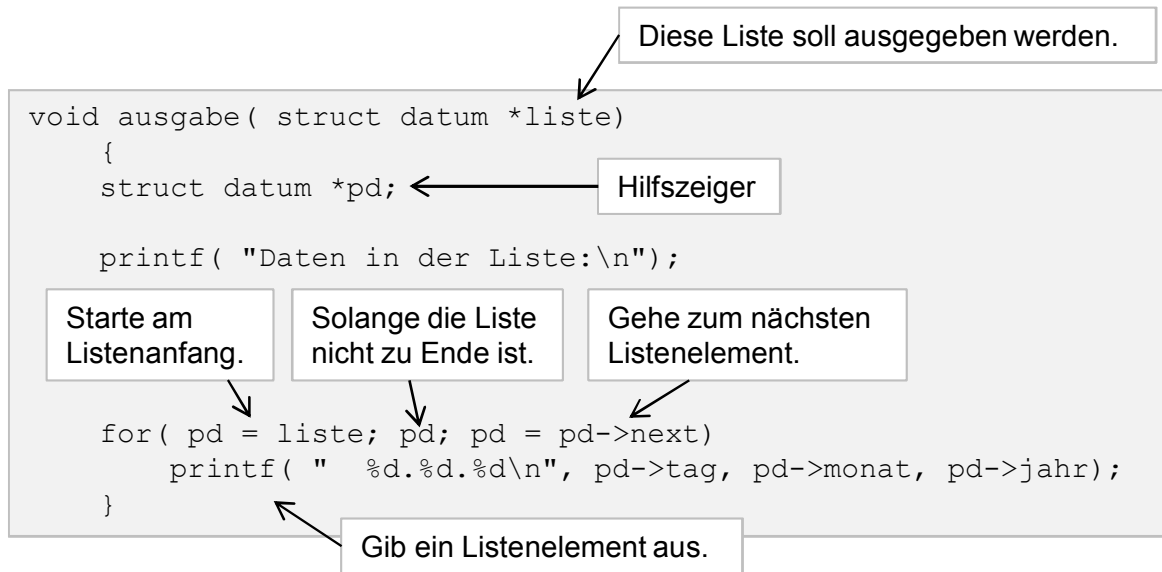


Der Listenanker wird zurückgegeben.

Die Funktion gibt einen Zeiger auf die erstellte Liste (Listenanker) an das rufende Programm zurück

Verwendung einer Liste

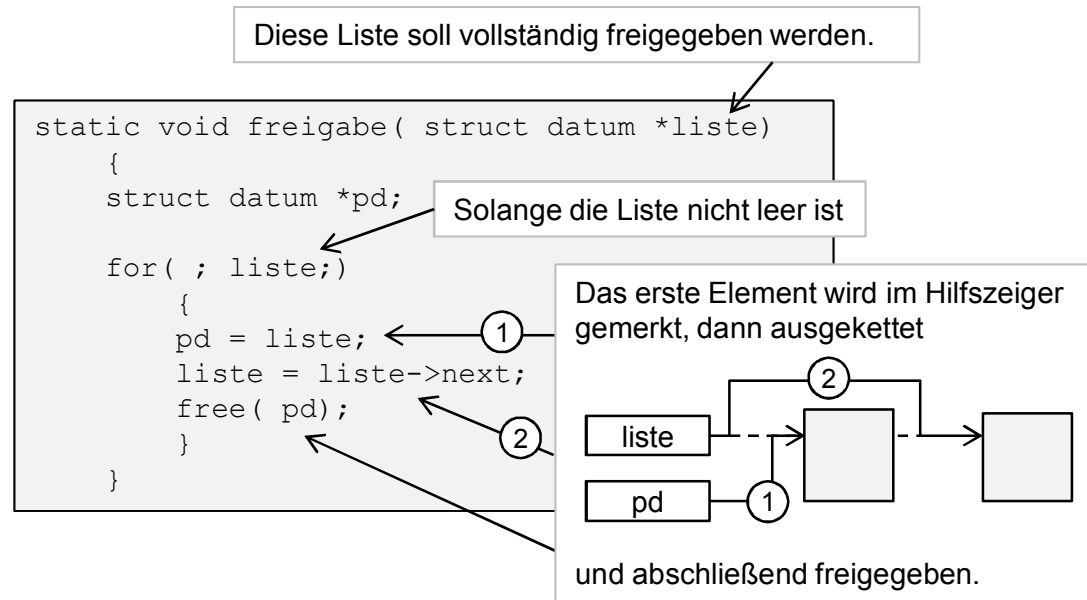
Die folgende Funktion iteriert über die Liste mit den Kalenderdaten und gibt die Listenelemente auf dem Bildschirm aus:



Das Ende der Liste wird am NULL-Zeiger erkannt.

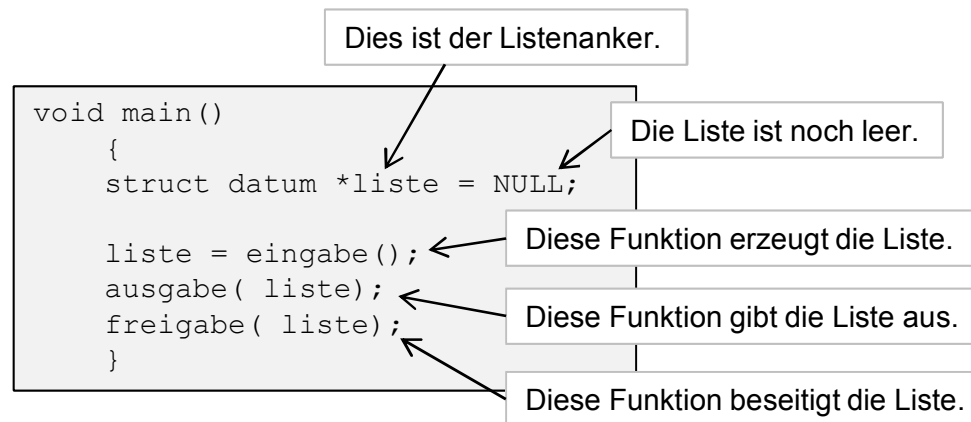
Freigabe einer Liste

Die Liste wird durch Ausketten des jeweils ersten Elements schrittweise in den Listenanker zurückgezogen, bis sie keine Elemente mehr hat.



Das ausgekettete Element wird mit `free` freigegeben.

Das komplette Programm

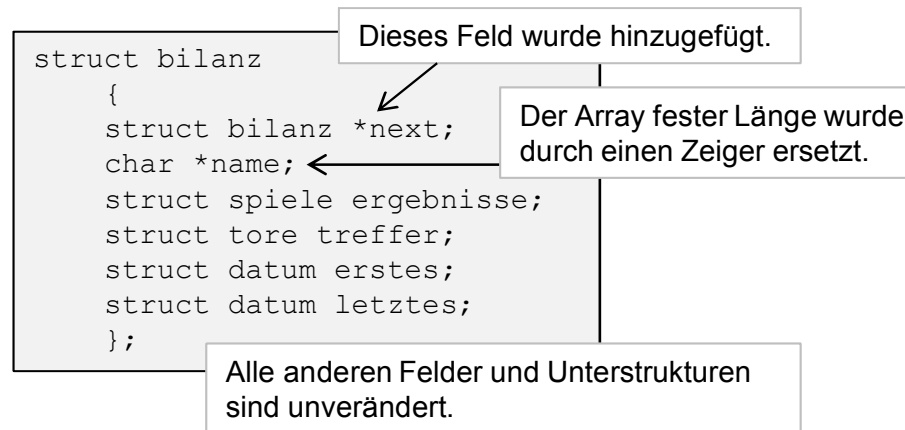


Jetzt entscheidet der Benutzer und nicht der Programmierer, wie viele Datensätze im Programm angelegt und verwaltet werden.

```
Noch ein Datum? 1
Datum: 29.12.1999
Noch ein Datum? 1
Datum: 30.12.1999
Noch ein Datum? 1
Datum: 31.12.1999
Noch ein Datum? 1
Datum: 1.1.2000
Noch ein Datum? 1
Datum: 2.1.2000
Noch ein Datum? 0
Daten in der Liste:
2.1.2000
1.1.2000
31.12.1999
30.12.1999
29.12.1999
```

Umstellung des Länderspielprogramms auf dynamische Datenstrukturen 1

Die Struktur `bilanz` wird so erweitert, dass Bilanzen als Liste gespeichert werden können. Dazu wird ein Zeiger in die Datenstruktur aufgenommen:

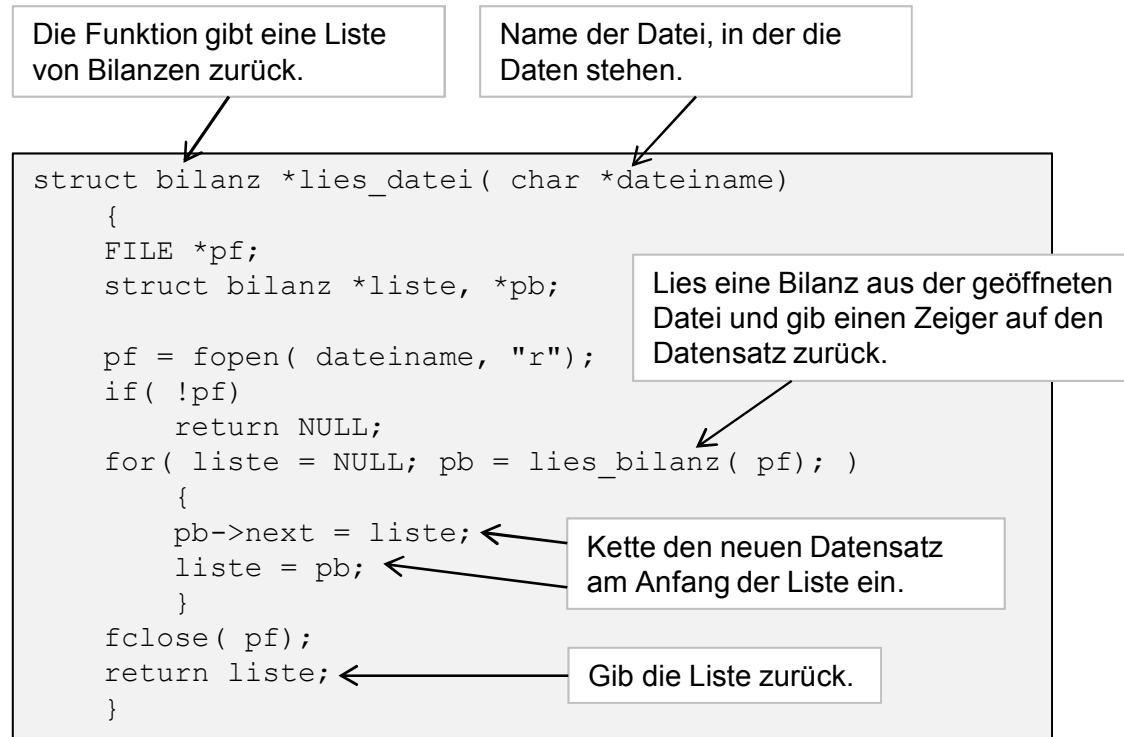


Es gibt aber noch eine wichtige Änderung:

Die Datenstruktur enthält jetzt nicht mehr den Speicher für den Ländernamen sondern nur noch einen Zeiger. Der Speicher für den Ländernamen muss getrennt allokiert werden. Dies gibt uns die Möglichkeit, nur so viel Speicher zu allokiieren, wie der Ländername exakt benötigt. Vorher war der Speicher auf eine maximal zu erwartende Länge ausgelegt.

Umstellung des Länderspielprogramms auf dynamische Datenstrukturen 2

Die Funktion `lies_datei` muss jetzt auf Listenverarbeitung umgestellt werden:



Durch das Einketten neuer Bilanzen am Anfang der Liste, wird die Reihenfolge der Bilanzen im Speicher gegenüber der Datei umgedreht. Das ist grundsätzlich kein Problem. Später werden wir eine Variante dieser Funktion betrachten, bei der die alphabetische Ordnung der Datei erhalten bleibt.

Umstellung des Länderspielprogramms auf dynamische Datenstrukturen 3

Beim Lesen einer einzelnen Bilanz muss jetzt der benötigte Speicher für den Datensatz und für den Ländernamen allokiert werden:

Zunächst wird versucht, einen weiteren Ländernamen aus der Datei zu lesen. Wenn das gelingt, wird eine neue Bilanz allokiert. Dann wird die Länge des Ländernamens festgestellt, entsprechend Speicher bereitgestellt und mit der Bilanz verlinkt. Nachdem der Ländername in den allokierten Speicher kopiert wurde, werden auch die restlichen Daten der Bilanz aus der Datei gelesen und in die Struktur geschrieben.

```
struct bilanz *lies_bilanz( FILE *pf)
{
    char land[100];
    struct bilanz *pb;

    fscanf( pf, "%s", land);
    if( feof( pf))
        return NULL;

    pb = (struct bilanz *)malloc( sizeof( struct bilanz));

    pb->name = (char *)malloc( strlen( land)+1);
    strcpy( pb->name, land);

    fscanf( pf, "%d %d %d %d", &pb->ergebnisse.gesamt,
                                &pb->ergebnisse.gew,
                                &pb->ergebnisse.unent,
                                &pb->ergebnisse.verl);

    fscanf( pf, "%d:%d", &pb->treffer.dfb,
                        &pb->treffer.gegner);
    fscanf( pf, "%d.%d.%d", &pb->erstes.tag,
                            &pb->erstes.monat,
                            &pb->erstes.jahr);
    fscanf( pf, "%d.%d.%d", &pb->letztes.tag,
                            &pb->letztes.monat,
                            &pb->letztes.jahr);

    return pb;
}
```

Zwischenspeicher für den Ländernamen. Versuchswises Lesen des Ländernamens.

Allokieren des Speichers für eine neue Bilanz.

Allokieren des Speichers für den Ländernamen. Kopieren des Namens.

Lesen des restlichen Datensatzes.

Rückgabe eines Zeigers auf den neuen Datensatz.

Umstellung des Länderspielprogramms auf dynamische Datenstrukturen 5

Einketten neuer Bilanzen am Ende der Liste:

```
struct bilanz *lies_datei2( char *dateiname)
{
    FILE *pf;
    struct bilanz *liste;
    struct bilanz **ppb;

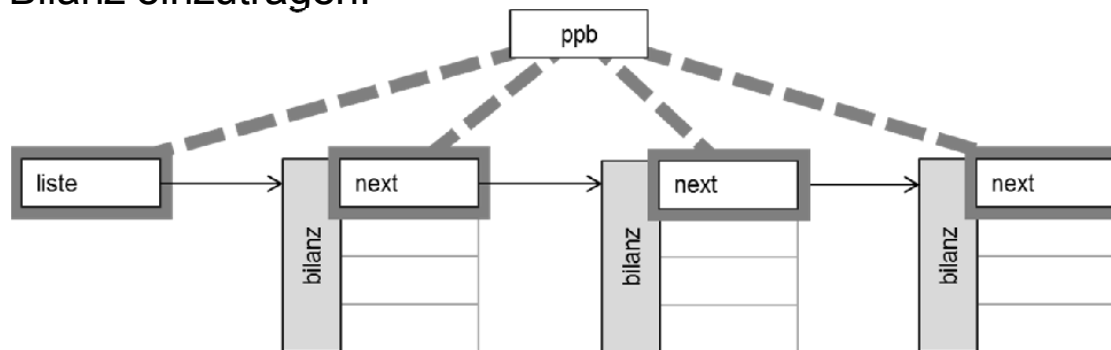
    pf = fopen( dateiname, "r");
    if( !pf)
        return NULL;
    for( ppb = &liste; *ppb = lies_bilanz( pf); ppb = &((*ppb)->next))
        ;
    fclose( pf);
    return liste;
}
```

Der Zeiger auf den Listenanfang.

Ein Zeiger auf einen Zeiger auf eine Bilanz.

Diese Schleife wird unten im Detail erklärt.

Der Zeiger `ppb` ist ein Zeiger auf ein Verkettungsfeld in der Liste (`struct bilanz **ppb`, doppelte Indirektion). Dieser Zeiger zeigt uns, wo das jeweils nächste Element der Liste einzuketten ist. Mit diesem Zeiger gehen wir von Verkettungsfeld zu Verkettungsfeld, bis wir an dem Verkettungsfeld angekommen sind, in dem eine 0 steht. Hier ist der Zeiger auf die neue Bilanz einzutragen:

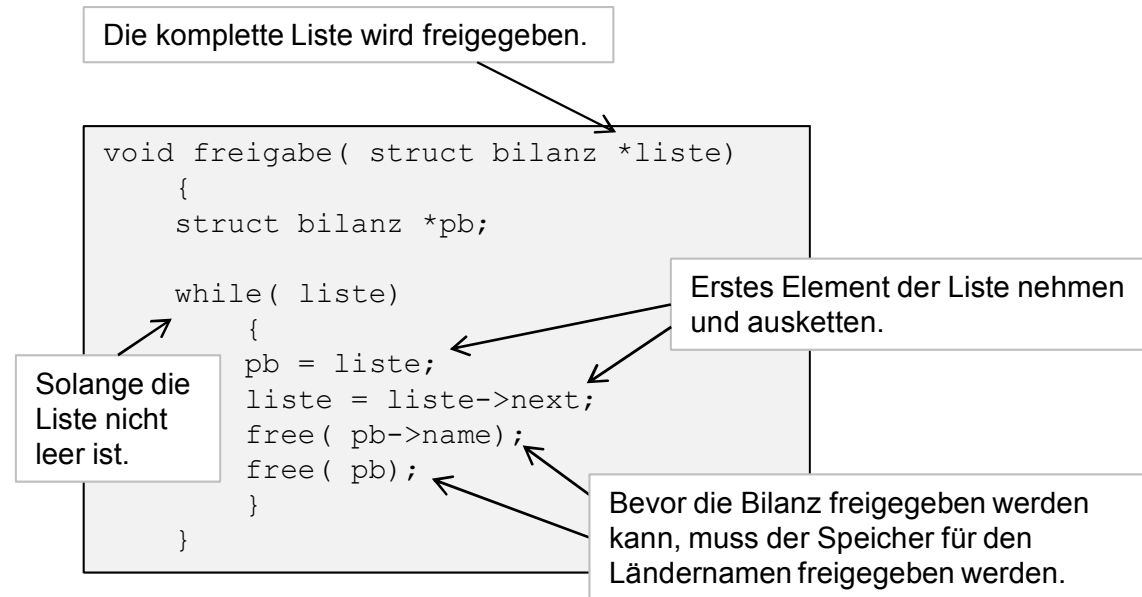


Effizienter wäre es, sich einen Zeiger auf das letzte Element der Liste zu merken, damit man die Liste nicht immer erneut durchlaufen muss. Das können Sie zur Übung selbst implementieren.

Ich habe dieses Vorgehen gewählt, da wir diese Technik später erneut verwenden werden und Sie sich schon einmal mit der doppelten Indirektion vertraut machen können.

Umstellung des Länderspielprogramms auf dynamische Datenstrukturen 4

Freigabe aller Daten:

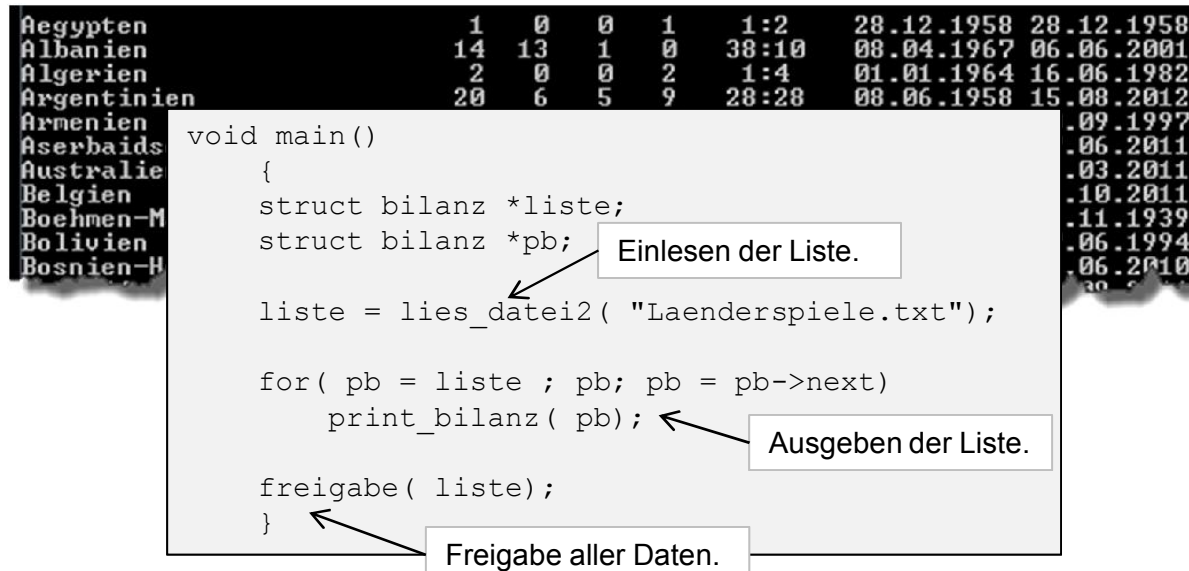


Die Freigabe einer Liste haben wir bereits früher einmal implementiert. Hier ist zusätzlich darauf zu achten, dass der Speicher für den Ländernamen freigegeben wird, bevor der Speicher für die Bilanz zurückgegeben wird.

Umstellung des Länderspielprogramms auf dynamische Datenstrukturen 5

Die Funktion `print_bilanz` zur Ausgabe einer Bilanz auf dem Bildschirm bleibt unverändert.

Im Hauptprogramm wird statt durch einen Array jetzt durch eine Liste iteriert:



```
void main()
{
    struct bilanz *liste;
    struct bilanz *pb;

    liste = lies_datei2( "Laenderspiele.txt");

    for( pb = liste ; pb; pb = pb->next)
        print_bilanz( pb);

    freigabe( liste);
}
```

Einlesen der Liste.

Ausgeben der Liste.

Freigabe aller Daten.

Land	1	0	0	1	1:2	28.12.1958	28.12.1958
Ägypten	14	13	1	0	38:10	08.04.1967	06.06.2001
Albanien	2	0	0	2	1:4	01.01.1964	16.06.1982
Argentinien	20	6	5	9	28:28	08.06.1958	15.08.2012
Armenien							09.1997
Aserbaids							06.2011
Australie							03.2011
Belgien							10.2011
Bohmen-M							11.1939
Bolivien							06.1994
Bosnien-H							06.2010

Suchen in Listen

Listen sind hervorragend geeignet, um darin sequentiell Daten zu suchen:

Suche in dieser Liste dieses Land.

```
struct bilanz *select_land( struct bilanz *pb, char *land)
{
    for( ; pb; pb = pb->next)
    {
        if( !strcmp( land, pb->name))
            return pb;
    }
    return NULL;
}
```

Iteriere über die Liste.

Falls die Namen übereinstimmen, gib einen Zeiger auf den gefundenen Datensatz zurück.

Land nicht gefunden.

```
void main()
{
    struct bilanz *liste;
    struct bilanz *pb;

    liste = lies_datei2( "Laenderspiele.txt");

    pb = select_land( liste, "Italien");
    if( pb)
        print_bilanz( pb);

    freigabe( liste);
}
```

Suche Daten für Italien.

Gib die Daten aus, falls das Land gefunden wurde.

Italien	31	7	9	15	35:47	01.01.1923	28.06.2012
---------	----	---	---	----	-------	------------	------------

Fortgesetztes Suchen in Listen

Suchen, bei denen man mehrere Treffer erwartet, lassen sich elegant durch Listen realisieren:

Suche in dieser Liste das erste Land, dessen Name mit diesem Buchstaben beginnt.

```
struct bilanz *select_buchstabe( struct bilanz *pb, char buchstabe)
{
    for( ; pb; pb = pb->next)
    {
        if( pb->name[0] == buchstabe)
            return pb;
    }
    return NULL;
}
```

Iteriere über die Liste.

Falls die Buchstaben übereinstimmen, gib einen Zeiger auf den gefundenen Datensatz zurück.

Land nicht gefunden.

```
void main()
{
    struct bilanz *liste;
    struct bilanz *pb;

    liste = lies_datei2( "Laenderspiele.txt");
}
```

Starte am Listenanfang.

Solange ein weiteres Land gefunden wird

Gehe zum nächsten Land.

```
for( pb = liste; pb = select_buchstabe( pb, 'B'); pb = pb->next)
    print_bilanz( pb);
```

Gib den Treffer aus.

```
freigabe( liste);
}
```

Belgien	25	20	1	4	58:26	16.05.1910	11.10.2011
Boehmen-Machren	1	0	1	0	4:4	12.11.1939	12.11.1939
Bolivien	1	1	0	0	1:0	17.06.1994	17.06.1994
Bosnien-Herzegowina	2	1	1	0	4:2	11.10.2002	03.06.2010
Brasilien	21	4	5	12	24:39	05.05.1963	10.08.2011
Bulgarien	21	16	2	3	56:24	20.10.1935	21.08.2002

Optimierung von Datenstrukturen

Zum Abschluss dieses Abschnitts möchte ich Ihnen noch einen Hinweis zur Speicheroptimierung von Datenstrukturen geben. Wir betrachten zwei Datenstrukturen, die sich nur in der Anordnung ihrer Datenfelder unterscheiden

```
struct test1
{
    char c1;
    long l1;
    char c2;
    long l2;
    char c3;
    long l3;
    char c4;
    long l4;
};
```

```
struct test2
{
    long l1;
    long l2;
    long l3;
    long l4;
    char c1;
    char c2;
    char c3;
    char c4;
};
```

Ein Größenvergleich liefert ein überraschendes Ergebnis

```
void main()
{
    printf( "test1: %d\n", sizeof(struct test1));
    printf( "test2: %d\n", sizeof(struct test2));
}
```

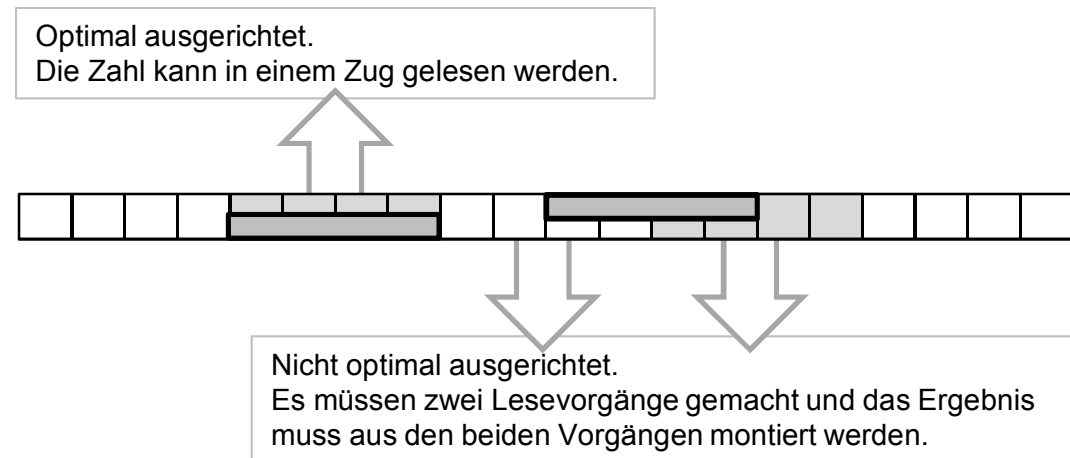
```
test1: 32
test2: 20
```

Die erste Struktur ist wesentlich (60%) größer als die zweite.

Woran liegt das?

Alignment

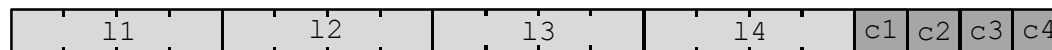
Intern werden Datenstrukturen so abgelegt, dass der Rechner optimal zugreifen kann. Wenn ein Rechner etwa eine 4-Byte Integer-Darstellung hat, greift er den Speicher in 4-Byte Blöcken ab und kann auf eine 4-Byte Zahl besonders effizient zugreifen, wenn sie auf einer durch 4 teilbaren Adresse beginnt.



Diese als **Alignment** bezeichnete Strategie zur Ausrichtung von Daten im Speicher führt dazu, dass im ersten Fall



sehr viel mehr Speicher benötigt wird als im zweiten.



Sie können ihre Datenstrukturen sehr einfach optimieren, indem Sie die Datenfelder absteigend nach Größe anordnen.

In der Programmlogik besteht kein Unterschied zwischen den beiden Varianten der Datenstruktur.

```
struct test1
{
    char c1;
    long l1;
    char c2;
    long l2;
    char c3;
    long l3;
    char c4;
    long l4;
};
```

```
struct test2
{
    long l1;
    long l2;
    long l3;
    long l4;
    char c1;
    char c2;
    char c3;
    char c4;
};
```