
Einführung in die Programmierung für Physiker

Die Programmiersprache C – Strukturen ("struct ...")

Marc Wagner

Institut für theoretische Physik
Johann Wolfgang Goethe-Universität Frankfurt am Main

SoSe 2024

Strukturen

- Eine **Struktur (structure)** ist eine Gruppierung mehrerer Variablen zu einem neuen Datentyp; die Variablen können unterschiedlichen Datentyps sein.
- Strukturen dienen der Organisation konzeptionell zusammengehöriger Variablen; sie vereinfachen die Programmierung und erhöhen die Lesbarkeit von Programmcode.
- Syntax (Definition einer Struktur):
 - `struct tag { type1 name1; type2 name2; ... };`
 - `tag` bezeichnet den Namen der Struktur (**Etikett, structure tag**).
 - `type1 name1` bezeichnet Datentyp und Namen der ersten Variable der Struktur (**Komponente, member**).
 - `type2 name2` bezeichnet Datentyp und Namen der zweiten Variable der Struktur (**Komponente, member**).
 - ...
- Nach einer solchen Strukturdefinition bildet `struct tag` einen eigenen Datentyp, kann syntaktisch also analog zu `int`, `double`, etc. verwendet werden; z.B. kann mit `struct tag name;` eine Variable vom Datentyp `struct tag` und mit Namen `name` definiert werden.
- Auf die Komponenten einer Struktur kann mit `.` zugegriffen werden, z.B. `name.name1`, `name.name2`, ...
- **Beispiel:** Struktur für 2-komponentige Vektoren und einige zugehörige Funktionen ...

```
1. #include<stdio.h>
2.
3. // *****
4.
5. // Definition einer Struktur fuer 2-komponentige Vektoren.
6. struct vec2d
7. {
8.     double x;
9.     double y;
10. };
11.
12. // *****
13.
14. // Initialisieren eines 2-komponentigen Vektors.
15. struct vec2d vec2d_init(double x, double y)
16. {
17.     struct vec2d v;
18.     v.x = x;
19.     v.y = y;
20.     return v;
21. }
22.
23. // *****
24.
25. // Ausgabe eines 2-komponentigen Vektors.
26. void vec2d_print(struct vec2d v)
27. {
28.     printf("( %.21f , %.21f )\n", v.x, v.y);
```

```

29. }
30.
31. // *****
32.
33. // Addition zweier 2-komponentiger Vektoren.
34. struct vec2d vec2d_add(struct vec2d v1, struct vec2d v2)
35. {
36.     struct vec2d v;
37.     v.x = v1.x + v2.x;
38.     v.y = v1.y + v2.y;
39.     return v;
40. }
41.
42. // *****
43.
44. // ... (Weitere Funktionen fuer 2-komponentige Vektoren.)
45.
46. // *****
47.
48. int main(void)
49. {
50.     struct vec2d v1, v2;
51.
52.     v1 = vec2d_init(1.0, 2.0);
53.     v2 = vec2d_init(3.0, 4.0);
54.
55.     printf("v1 = ");
56.     vec2d_print(v1);
57.     printf("v2 = ");
58.     vec2d_print(v2);
59.
60.     struct vec2d v3 = vec2d_add(v1, v2);
61.     printf("v3 = v1 + v2 = ");
62.     vec2d_print(v3);
63. }

```

```

v1 = ( +1.00 , +2.00 )
v2 = ( +3.00 , +4.00 )
v3 = v1 + v2 = ( +4.00 , +6.00 )

```

- Strukturen können nicht mit Vergleichsoperatoren verglichen werden.

```

43. ...
44. int main(void)
45. {
46.     struct vec2d v1, v2;
47.
48.     v1 = vec2d_init(1.0, 2.0);
49.     v2 = vec2d_init(3.0, 4.0);
50.
51.     if(v1 == v2) // Liefert einen Fehler beim Kompilieren.
52.         printf("v1 ist gleich v2.\n");
53.     else
54.         printf("v1 ist ungleich v2.\n");
55. }

```

```

mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ g++ -o prog prog.c
prog.c: In Funktion "int main()":
prog.c:60:12: Fehler: keine Übereinstimmung für "operator==" in "v1 == v2"
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ gcc -o prog prog.c
prog.c: In Funktion "main":
prog.c:60:9: Fehler: Ungültige Operanden für binäres == (haben "struct vec2d" und "struct vec2d")

```

- Ausweg: Benutze eine selbstgeschriebene "Vergleichsfunktion" an Stelle des Vergleichsoperators `==`.

```
43. ...
44. // Vergleich (==) zweier 2-komponentiger Vektoren.
45. int vec2d_is_eq(struct vec2d v1, struct vec2d v2)
46. {
47.     return (v1.x == v2.x) && (v1.y == v2.y);
48. }
49.
50. // *****
51.
52. int main(void)
53. {
54.     struct vec2d v1, v2;
55.
56.     v1 = vec2d_init(1.0, 2.0);
57.     v2 = vec2d_init(3.0, 4.0);
58.
59.     if(vec2d_is_eq(v1, v2))
60.         printf("v1 ist gleich v2.\n");
61.     else
62.         printf("v1 ist ungleich v2.\n");
63. }
```

v1 ist ungleich v2.

- Werden Zeiger auf Strukturen verwendet, kann auch mit `->` auf die Komponenten zugegriffen werden.

```
51. ...
52. // Vertauschen zweier 2-komponentiger Vektoren.
53. void vec2d_swap(struct vec2d *pv1, struct vec2d *pv2)
54. {
55.     double tmp;
56.
57.     tmp = pv1->x; // Aequivalent zu "tmp = (*pv1).x;", nicht aber zu "tmp = *pv1.x;".
58.     pv1->x = pv2->x;
59.     pv2->x = tmp;
60.
61.     tmp = pv1->y;
62.     pv1->y = pv2->y;
63.     pv2->y = tmp;
64. }
65.
66. // *****
67.
68. int main(void)
69. {
70.     struct vec2d v1, v2;
71.
72.     v1 = vec2d_init(1.0, 2.0);
73.     v2 = vec2d_init(3.0, 4.0);
74.
75.     vec2d_swap(&v1, &v2);
76.
77.     printf("v1 = ");
78.     vec2d_print(v1);
79.     printf("v2 = ");
80.     vec2d_print(v2);
```

```
v1 = ( +3.00 , +4.00 )
v2 = ( +1.00 , +2.00 )
```

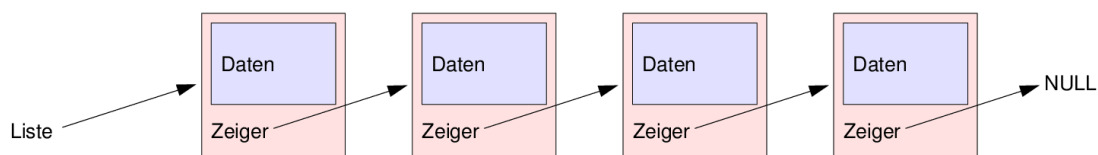
- Mit **typedef** *type name*; kann dem Datentyp *type* der neue Datentypname *name* zugewiesen werden; besonders bei Strukturen bietet sich die Definition eines kürzeren äquivalenten Datentypnamens an.

```
1. #include<stdio.h>
2.
3. // *****
4.
5. // Definition einer Struktur fuer einen 2-komponentigen Vektor.
6. struct vec2d
7. {
8.     double x;
9.     double y;
10. };
11.
12. typedef struct vec2d VEC2D; // VEC2D ist nun aequivalent zu struct vec2d.
13.
14. // *****
15.
16. // Initialisieren eines 2-komponentigen Vektors.
17. VEC2D vec2d_init(double x, double y)
18. {
19.     VEC2D v;
20.     v.x = x;
21.     v.y = y;
22.     return v;
23. }
24.
25. // *****
26.
27. // Ausgabe eines 2-komponentigen Vektors.
28. void vec2d_print(VEC2D v)
29. {
30.     printf("( %+2lf , %+2lf )\n", v.x, v.y);
31. }
32.
33. // *****
34.
35. // Addition zweier 2-komponentiger Vektoren.
36. VEC2D vec2d_add(VEC2D v1, VEC2D v2)
37. {
38.     VEC2D v;
39.     v.x = v1.x + v2.x;
40.     v.y = v1.y + v2.y;
41.     return v;
42. }
43.
44. // *****
45.
46. // Vergleich (==) zweier 2-komponentiger Vektoren.
47. int vec2d_is_eq(VEC2D v1, VEC2D v2)
48. {
49.     return (v1.x == v2.x) && (v1.y == v2.y);
50. }
51.
52. // *****
```

```
53.
54. // Vertauschen zweier 2-komponentiger Vektoren.
55. void vec2d_swap(VEC2D *pv1, VEC2D *pv2)
56. {
57.     double tmp;
58.
59.     tmp = pv1->x; // Aequivalent zu "tmp = (*pv1).x;", nicht aber zu "tmp = *pv1.x;".
60.     pv1->x = pv2->x;
61.     pv2->x = tmp;
62.
63.     tmp = pv1->y;
64.     pv1->y = pv2->y;
65.     pv2->y = tmp;
66. }
67.
68. // *****
69.
70. int main(void)
71. {
72.     VEC2D v1, v2;
73.
74.     v1 = vec2d_init(1.0, 2.0);
75.     v2 = vec2d_init(3.0, 4.0);
76.
77.     vec2d_swap(&v1, &v2);
78.
79.     printf("v1 = ");
80.     vec2d_print(v1);
81.     printf("v2 = ");
82.     vec2d_print(v2);
83. }
```

Anwendung: Einfach verkettete Listen

- Zur Verwaltung einer zunächst unbekannt Menge von Daten (z.B. Namen, Messergebnisse, etc.) ist ein statisches Array ungeeignet.
- Auch ein dynamisch angelegtes Array ist nicht ideal, wenn z.B. häufig Daten hinzugefügt werden; dann müsste jedes Mal ein entsprechend vergrößerter Speicherbereich angelegt werden und die existierenden Daten müssten in diesen neuen Speicherbereich kopiert werden (**sehr ineffizient**).
- Häufig werden für solche Probleme Listen eingesetzt.
- **Einfach verkettete Liste:**
 - Jedes Listenelement entspricht einer Struktur, die die Daten eines Elements (z.B. bei einem Messergebnis den Messwert, das Datum, eine Bemerkung, etc.) enthält, sowie einen Zeiger auf das nächste Listenelement.
 - Der Zeiger des letzten Listenelements besitzt den Wert **NULL**.
 - Die Liste entspricht einem Zeiger auf deren erstes Element; über den Zeiger des ersten Elements kann dann auf das Zweite zugegriffen werden, über den Zeiger des zweiten Elements kann dann auf das Dritte zugegriffen werden, etc.
 - Eine leere Liste ist ein Zeiger auf ein Listenelement, der den Wert **NULL** besitzt.
- Das Einfügen eines neuen bzw. Löschen eines existierenden Listenelements ist sehr effizient; lediglich Speicherplatz für ein Element muss dynamisch angefordert bzw. freigegeben werden.



```
1. #include<stdio.h>
2. #include<stdlib.h>
3. #include<string.h>
4.
5. // *****
6.
7. // Definition einer Struktur fuer ein Listenelement (enthaelt einen Namen).
8. struct l_elem
9. {
10.     char name[100];
11.     struct l_elem *next; // Zeiger auf das naechste Listenelement.
12. };
13.
14. typedef struct l_elem L_ELEM;
15.
16. // *****
17.
18. // Haengt ein neues Listenelement am Ende der Liste *p_list an; initialisiert
19. // das neue Listenelement mit name.
20. void l_append(L_ELEM **p_list, const char name[])
21. {
22.     // Dynamisches Anlegen des neuen Listenelements.
23.
24.     L_ELEM *p_new_elem;
25.
```

```

26.  if((p_new_elem = (L_ELEM *)malloc(sizeof(L_ELEM))) == NULL)
27.  {
28.      printf("Fehler in void l_append(...\n");
29.      exit(0);
30.  }
31.
32.  strcpy(p_new_elem->name, name);
33.  p_new_elem->next = NULL;
34.
35.  // *****
36.
37.  // Anhaengen des neuen Listenelements an die bestehende Liste.
38.
39.  if(*p_list == NULL)
40.      // Die bestehende Liste ist leer
41.      // --> setze die Liste gleich dem neuen Element.
42.      {
43.          *p_list = p_new_elem;
44.          return;
45.      }
46.
47.  // Die bestehende Liste ist nicht leer
48.  // --> suche das Ende der Liste und haenge das neue Element an.
49.
50.  L_ELEM *p_tmp = *p_list;
51.
52.  while(p_tmp->next != NULL)
53.      p_tmp = p_tmp->next;
54.
55.  p_tmp->next = p_new_elem;
56. }
57.
58. // *****
59.
60. // Ausgabe der Liste p_list.
61. void l_print(L_ELEM *p_list)
62. {
63.     if(p_list == NULL)
64.         return;
65.
66.     printf("%s\n", p_list->name);
67.     l_print(p_list->next);
68. }
69.
70. // *****
71.
72. // Loescht das index-te Element aus der Liste.
73. void l_delete(L_ELEM **p_list, int index)
74. {
75.     int i1;
76.
77.     if(index == 0)
78.         // Loeschen des 0-ten Listenelements (index == 0).
79.         {
80.             if(*p_list == NULL)
81.                 {
82.                     printf("Fehler in void l_delete(...\n");
83.                     exit(0);
84.                 }

```



```

85.
86.     L_ELEM *p_del = *p_list; // Das zu loeschende Element.
87.     *p_list = (*p_list)->next; // Die Liste wird zu einem Zeiger auf das 1-te Listenelement.
88.     free(p_del); // Freigeben des Speichers.
89. }
90. else
91.     // Loeschen eines Listenelements, das nicht am Anfang der Liste steht (index >= 1).
92.     {
93.         L_ELEM *p_tmp = *p_list;
94.
95.         for(i1 = 0; i1 < index-1; i1++)
96.             {
97.                 if(p_tmp == NULL)
98.                     {
99.                         printf("Fehler in void l_delete(...\n");
100.                        exit(0);
101.                    }
102.
103.                    p_tmp = p_tmp->next;
104.                }
105.
106.                // p_tmp zeigt jetzt auf das Listenelement vor dem zu loeschenden Element.
107.
108.                if(p_tmp->next == NULL)
109.                    {
110.                        printf("Fehler in void l_delete(...\n");
111.                        exit(0);
112.                    }
113.
114.                    L_ELEM *p_del = p_tmp->next; // Das zu loeschende Element.
115.                    p_tmp->next = p_tmp->next->next; // Das zu loeschende Element wird "uebersprungen".
116.                    free(p_del); // Freigeben des Speichers.
117.                }
118.    }
119.
120. // *****
121.
122. int main(void)
123. {
124.     L_ELEM *list = NULL; // Legt eine leere Liste an.
125.
126.     l_append(&list, "Donald");
127.     l_append(&list, "Dagobert");
128.     l_append(&list, "Tick");
129.     l_append(&list, "Trick");
130.     l_append(&list, "Track");
131.     l_print(list);
132.     printf("*****\n");
133.
134.     l_delete(&list, 3);
135.     l_print(list);
136.     printf("*****\n");
137.
138.     l_delete(&list, 0);
139.     l_print(list);
140.     printf("*****\n");
141.
142.     l_delete(&list, 2);
143.     l_print(list);

```

```

144. printf("*****\n");
145.
146. l_append(&list, "Pluto");
147. l_append(&list, "Goofy");
148. l_print(list);
149. printf("*****\n");
150.
151. l_delete(&list, 4);
152. }

```

```

Donald
Dagobert
Tick
Trick
Track
*****
Donald
Dagobert
Tick
Track
*****
Dagobert
Tick
Track
*****
Dagobert
Tick
*****
Dagobert
Tick
Pluto
Goofy
*****
Fehler in void l_delete(...)

```

- **Hausaufgabe:** Schreibe weitere Funktionen für einfach verkettete Listen, z.B.
 - Einfügen eines neuen Elements an einer bestimmten Position,
 - Kopieren einer Liste,
 - Umdrehen einer Liste,
 - ...