Proseminar „C-Programmierung“

Strukturen

Von Marcel Lebek

Index

Was sind Strukturen?

Strukturen sind Zusammenschlüsse mehrerer elementarer Datentypen (Integer, Char, Boolean, …), die unter einem neuen Datentyp zusammengefasst werden. Deklariert wird eine Struktur mit dem Begriff *struct* nach folgendem Schema:

```
struct Strukturname
{
int a;
bool b;
char c[1];
};
```

Das Erzeugen einer Struktur ist auch direkt bei der Deklaration möglich:

```
struct Strukturname
{
int a;
bool b;
char c[1];
}struk1, struk2={1, 0, 'e'};
```

Zu erwähnen ist, dass am Ende einer Strukturdeklaration ein Semikolon steht und man beim Erzeugen einer Struktur immer das Schlüsselwort `struct` vor den Strukturnamen schreiben muss.

Um Fehler beim Kompilieren zu vermeiden, ist es auch möglich den Strukturnamen ohne Variablen zu Beginn zu deklarieren und den genauen Inhalt der Struktur später hinzuzufügen:

```
struct Strukturname;
```

Dies ist vor Allem nötig, wenn Strukturen gegenseitig aufeinander zugreifen.

Beim Erzeugen von Strukturen kann man bereits nach folgendem Schema den internen Variablen Werte zuweisen:

```
struct Strukturname struk1 = {1, 0, 'r'};
```

Der Zugriff auf die einzelnen Elemente erfolgt durch Punktnotation, womit jedes Element einzeln angesprochen werden kann:

```
struk1.a = 1;
```

Das Gruppieren von zusammenhängenden Variablen ist natürlich wesentlich übersichtlicher, das ist jedoch nicht der Grund, dass es Strukturen gibt. Während C keine Klassen unterstützt, die die objektorientierte Programmierung ermöglichen, bieten Strukturen trotzdem bedingt die Möglichkeit, objektorientiert zu programmieren. Dies wird besonders deutlich, wenn man sich den objektorientierten Nachfolger von C, C++ ansieht. Dieser hat das Prinzip der Strukturen um klassentypische Elemente wie Polymorphie und Vererbung, sowie der Möglichkeit der `public/private` Deklaration von Variablen oder der Objektspezifischen Funktionen, erweitert und unter dem Typ `class` vereint. All diese Dinge braucht man jedoch nicht zwingend, um kleine Schritte in der objektorientierten Programmierung zu machen, das ermöglichen Strukturen also. Im späteren Verlauf wird darauf noch einmal genauer anhand eines Beispiels eingegangen.

Im Speicher werden alle Strukturelemente in der Reihenfolge ihrer Deklaration angeordnet. Neuere Compiler jedoch fügen mehrere Char-Variablen zu einer großen zusammen, sodass die Reihenfolge intern unter Umständen unterschiedlich geregelt wird.

Hier zum Schluss noch ein Beispiel zu Strukturen und ihrer Handhabung:

```
struct Person
{
char vorname[24];
char nachname[24];
int alter;
bool verheiratet;
};

struct Person marcel={'Marcel', 'Lebek', 22, 0};

struct Person peter;
peter.vorname = ”Peter”;
peter.Nachname = ”Pan”;
peter.alter = 43;
peter.verheiratet = 0;
```

Padding

Das folgende Programm kommt nach Ausführung zum gezeigten Ergebnis:

```
#include <stdio.h>

struct Struktur1
{
  bool a;
  char b[6];
  long double c;

} Struk1;

struct Struktur2
{
  bool a;
  long double c;
  char b[6];

} Struk2;


int main()
{
      printf("Laenge von Struk1 (bool-long double-char[6]: %i\n",
sizeof(Struk1));
      printf("Laenge von Struk2 (bool-char[6]-long double: %i",
sizeof(Struk2));
      getchar();
}
```

```
Laenge von Struk1 (bool-long double-char[6]: 16

Laenge von Struk2 (bool-long double-char[6]: 24
```

Wie man sieht beinhalten beide Strukturen dieselben Variablen, lediglich die Reihenfolge ist unterschiedlich. Dennoch ist die erste Struktur 16 Byte groß, während die zweite 24 Bytes groß ist.

Dieser Effekt tritt durch sogenanntes “Padding“ auf. Hierbei füllt der Compiler per Präprozessorbefehl leere Bytes zwischen Variablen, um einen schnelleren Zugriff zu ermöglichen. An dieser Stelle sei noch erwähnt, dass dies Compilerbedingt verschieden gelöst wird. Ebenso entstehen Unterschiede zwischen 32- und 64-bit-Versionen des Betriebssystems. Die hier gezeigten Beispiele wurden mit dem Microsoft Visual C++ Compiler unter 64 bit getestet.

Das Padding wird eingesetzt, um 2 Bedingungen zu erfüllen:

1. **Die Adresse, an der eine Variable beginnt, ist ein vielfaches Ihrer Größe**

   Dies bedeutet, dass vor jeder Variable abhängig von ihrer Größe bis zu *`sizeof(Variable)-1`* Padding-Bytes eingefügt werden:

Datentyp	**Länge**	**Mögliches Padding im ungünstigsten Fall**
bool	1	0
char	1	0
short	2	1
int	4	3
float	4	3
long	4	3
long double	8	7

   So kommt es, dass in folgendem Beispiel ganze 7 Padding-Bytes eingefügt werden:

```
struct Struktur3
{
  bool a;
  long double c;
};
```

   Schaut man sich diese Struktur auf dem Stack an, würde sie in etwa so aussehen:

Adresse	Variable im Speicher
0x0000	bool a
0x0001	char padding[7];
0x0008	long double c;

   Wie man sieht ist die Adresse von *c* genau ein Vielfaches von der Typgröße `long double`.

2. **Die Strukturgröße ist ein Vielfaches der Größe ihrer größten Variable**

   Sollte eine Struktur samt all ihrer Padding-Bytes aus Bedingung 1 nicht die n-fache Größe ihrer größten Variable haben, so werden so viele Bytes aufgefüllt, bis diese Bedingung ebenfalls gegeben ist.

```
struct Struktur4
{
  int i
  char b[1];
};
```

   Diese Struktur besitzt die Länge 8. Padding nach Bedingung 1 ist nicht nötig, da Variablen vom Typ char immer ohne Padding auskommen und vor der ersten Variable ohnehin kein Padding angewendet wird. Die Strukturlänge wäre nun jedoch 5, was kein n-faches vom größten Variablentyp – dem Integer, 4 – ist. Durch Einfügen von 3 Padding-Bytes erfüllen wir die zweite Bedingung und haben eine korrekt organisierte Struktur.

Vor- und Nachteile von Padding

Padding verbessert die Laufzeit auf Kosten von Speicher. Das Ganze mag bei kleineren Programmen wenig ausmachen. Bei begrenztem Speicher jedoch kann dies zu einem Problem werden, wie in folgendem Beispiel gezeigt wird:

```
struct Auto
{
  bool a;
  long double kennungsnummer;
};

struct Auto autoliste[10];
```

Das Array Autoliste hat nun die Größe von 160 Byte, effektiv genutzt davon werden aber nur 90 Byte. Das ist ein Verbrauch von fast 44% mehr als nötig. Bei einem Array mit 10 Feldern wirkt das relativ unscheinbar, aber bei 54 Millionen Autos in Deutschland ist der Unterschied theoretisch zwischen 8,2 GB und 4,6 GB schon eher bemerkbar.

Die Wahl, ob man Padding nutzt oder nicht, hat man jedoch auch nur bedingt. Einige Compiler bieten Präprozessorbefehle an, die das Nutzen von Padding untersagen. Dies kann aber zu Compilerfehlern oder sogar Programmabstürzen führen, je nach Compiler und System verschieden. Unter dem Microsoft Visual C++ Compiler gibt es die Möglichkeit, eine Struktur mit dem Schlüsselwort UNALIGNED als „nicht korrekt strukturierte Struktur“ zu kennzeichnen. Diese Strukturen überprüft der Compiler dann auf Korrektheit und bessert notfalls nach. Die erhoffte Platzersparnis bleibt dann jedoch aus.

Aber es gibt nicht nur die beiden Extreme, entweder Platz oder Laufzeit sparen, denn in vielen Fällen lässt sich schon ein Teil der Paddingbytes durch simples umstrukturieren der Strukturvariablen verhindern.

Padding gering halten

Um möglichst wenig Padding innerhalb einer Struktur zu haben, ist es wichtig die Reihenfolge der Strukturvariablen zu beachten. Man muss darauf achten, dass man Stellen, an denen Padding auftreten könnte, möglichst mit passenden anderen Strukturvariablen füllt.

```
struct Struktur5
{
  bool a;
  long double c;
  char b[3];
};
```

Diese Struktur sieht nach dem Kompilieren wie folgt aus und kommt auf eine Länge von 16 Byte:

```
struct Struktur5
{
  bool a;
  char padding1[3];
  long double c;
  char b[3];
  char padding2[1];
};
```

Bei genauerem Hinsehen merkt man, dass nun 3 Padding-Bytes an einer Stelle eingefügt wurden. Gleichzeitig existiert innerhalb der Struktur eine gleichgroße Strukturvariable, die man an dieser Stelle hätte einsetzen können. Wenn man jetzt genau dies tut, verkleinert sich die Struktur auf 12 Byte.

Vor dem Kompilieren:

```
 struct Struktur6
{
  bool a;
  char b[3];
  long double c;
};
```

Nach dem Kompilieren:

```
struct Struktur6
{
  bool a;
  char b[3];
  long double c;
};
```

Ein optimales Ergebnis erhält man in der Regel dann, wenn man die Strukturvariablen nach ihrer Größe geordnet, beginnend mit der Größten, deklariert. Mit dieser Methode lässt sich Bedingung 1 für das Auftreten von Padding komplett ausschließen. Lediglich am Ende einer Struktur müssen eventuell noch Padding-Bytes eingefügt werden, diese lassen sich allerdings nicht verhindern. Dennoch ist es mit dieser Methode möglich, einiges an Speicherplatz einzusparen.

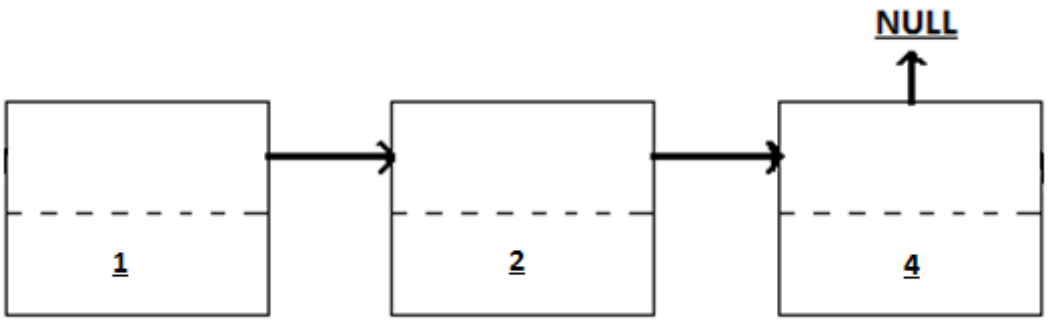
Anwendungsgebiete von Strukturen

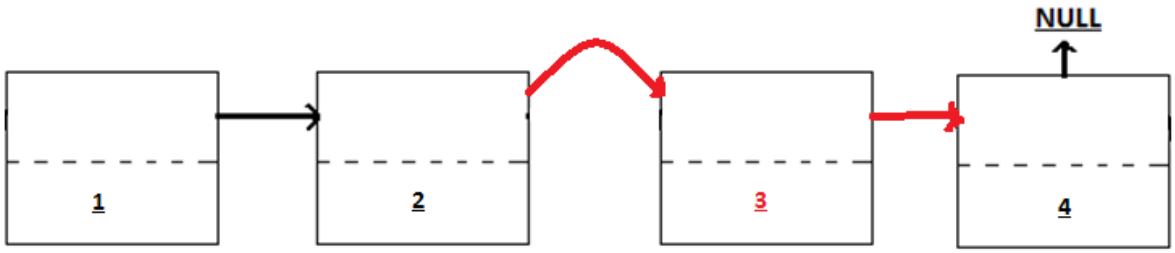
Eines der nützlichsten Anwendungsgebiete von Strukturen ist das Erzeugen von Listen und Bäumen. Dadurch, dass man eine Variable und einen bzw. mehrere Pointer unter einer Struktur vereinen kann, ist es möglich, Listen- und Baumelemente zu erzeugen:

```
struct Listenelement {
 int wert;
 struct Listenelement *nachfolger;
};
```

Erzeugt man ein Listenelement, kann man nun ein weiteres Element erzeugen und den Pointer des ersten Elementes auf dieses setzen. So lassen sich beliebig lange Listen erzeugen.

Nun könnte man so eine Liste auch in einem Array gestalten, sofern man eine feste Anzahl der Listenelemente hat. Doch auch dieses Vorgehen birgt immer noch Nachteile im Vergleich zur Struktur, da man wesentlich mehr Aufwand für das Einfügen eines Listenelementes an einer beliebigen Stelle benötigt. Beim Array muss man dann jedes Element hinter dem Neuen verschieben, also umschreiben. Bei der Struktur-Liste muss man lediglich zwei Pointer ändern.





Unions

Unions ähneln den Strukturen sehr, werden allerdings völlig unterschiedlich genutzt. Eine Union ist wie die Struktur ein Zusammenschluss mehrerer elementarer Datentypen. Auch von der Schreibweise unterscheiden sie sich nur durch das Schlüsselwort `union`:

```
union Union1
{
  int a;
  float b;
};
```

Der Unterschied zu Strukturen ist jedoch, dass eine Union nur Speicherplatz für die größte Variable innerhalb der Union einplant. Dieser Speicher wird für sämtliche Union-Variablen genutzt, es kann also immer nur ein Wert in einer Union gespeichert werden. Der Nutzen liegt darin, dass man den gespeicherten Wert hierdurch als verschiedene Datentypen interpretieren kann.

Würde man eine Union nach obigem Schema erzeugen und einmal die Variable *a = 1* und einmal *b = 1,2345* setzen, um danach jeweils a und b auszulesen, bekäme man folgende Ergebnisse:

```
#include <stdio.h>

int main()
      {
        union
        {
          int a;
          float b;
        } zahlen;



      zahlen.a = 1;
      printf("1. int: %i \n",zahlen.a);
      printf("1. float: %fl \n",zahlen.b);

      zahlen.b = 1.2345;
      printf("2. int: %i \n",zahlen.a);
      printf("2. float: %fl \n",zahlen.b);
      while(1) { };
      }
```

1. int: 1
1. float: 0.0000001
2. int: 1067320345
2. float: 1.2345001

Bevor man dieses Ergebnis analysiert ist es wichtig, den Aufbau der beiden Datentypen zu kennen. Ein Integer wird einfach in Binärschreibweise in den Speicher geschrieben. Ein Float hingegen wird als Kombination aus Mantisse und Exponent als Gleitkommazahl dargestellt.

1 bit	7 Bit	24 Bit
Vorzeichen	Exponent	Mantisse

Im ersten Durchlauf steht nun der Integerwert 1 in der Union. Die Ausgabe des Integers ist natürlich gleich, die Ausgabe des Floats hingegen interpretiert nun den Integerwert als Float, was eben nicht 1, sondern 0,0000001 ist. Beim zweiten Durchlauf ist es andersherum, hier ist der Float 1,2345 in der Union gespeichert. Der Integer liest nun den hinteren Teil des Floats aus und interpretiert diesen als Integer, deshalb kommt dabei letztendlich nur ein Teilbereich des Floats als Ausgabe heraus.

Nun stellt man sich die Frage, wie sinnvoll es ist einen falsch interpretierten Wert auszulesen. In einer Situation jedoch, in der man den erwarteten Datentyp nicht hundertprozentig voraussehen kann, ist eine Union die Lösung aller Probleme. Sie ermöglicht es, mehrere verschiedene Datentypen innerhalb einer Union interpretieren zu können.

Quellen

http://en.wikipedia.org/wiki/Struct_(C_programming_language)

http://en.wikipedia.org/wiki/Data_structure_alignment

http://msdn.microsoft.com/en-us/library/ms253935(v=VS.90).aspx

http://home.htw-berlin.de/~junghans/cref/SYNTAX/struct.html

http://de.wikibooks.org/wiki/C-Programmierung

http://stackoverflow.com/questions/2748995/c-struct-memory-layout

http://www.c-howto.de

Jürgen Wolf, „C von A bis Z“

Helmut O.B. Schellong, „Moderne C-Programmierung“