Strukturen, Unions, Bitfelder

Einführung in die Programmierung

Michael Felderer (QE)

Institut für Informatik, Universität Innsbruck

STRUKTUREN

Allgemein

- Eine Struktur ist eine Ansammlung von mehreren Variablen (möglicherweise mit **unterschiedlichen** Typen), die unter einem **einzigen Namen** zusammengefasst werden.
- Mit einer Struktur kann man daher einen eigenen Typ definieren und dann im Programm bei Deklarationen verwenden.
- Eine Struktur dient zur Organisation der Daten.

Deklaration und Definition

- Eine Deklaration bzw. Definition fängt immer mit dem Schlüsselwort `struct` an und enthält eine Liste von Elementen in geschweiften Klammern.
  - Die Namen der Elemente einer Struktur müssen eindeutig sein.
- Beispiele für einen Punkt (x- und y-Koordinaten):

getrennt	zusammengefasst

`struct point {`
`  int x;`
`  int y;`
`};`
 	
`struct point pt;`
	`struct point {`
	`  int x;`
	`  int y;`
	`} pt;`
	
	`oder`
	
	`struct {`
	`  int x;`
	`  int y;`
	`} pt;`

Zugriff

- Um auf einzelne Elemente einer Strukturvariable zugreifen zu können, wird der Punkt-Operator „.“ verwendet.
  - Der Zugriff erfolgt aber sonst wie bei normalen Variablen.
- Beispiel bei `struct point`:
  - Zugriff auf Komponente x bzw. y
  - `pt.x + pt.y …`

Initialisierung

- Die Initialisierung erfolgt mit einer Initialisierungsliste.
- Beispiel

```
struct point maxpt = { 320, 200 };
```

- In der Liste können auch einzelne Initialisierer (am Ende) freigelassen werden.
  - Diese werden dann automatisch initialisiert, falls die Struktur global oder statisch deklariert wird.
  - Ansonsten haben die einzelnen Elemente einen undefinierten Wert!
- Für die Initialisierung einer Strukturvariable darf auch eine andere Strukturvariable verwendet werden (Zuweisung).
- Im C99-Standard ist es möglich, nur bestimmte Elemente einer Struktur in beliebiger Reihenfolge zu initialisieren.
  - Dabei werden die einzelnen Elemente über ihren Namen angesprochen.
  - Beispiel (siehe übernächste Folie).

Operationen auf Strukturen

- Folgende Operationen sind laut ANSI C auf Strukturen erlaubt:
  - Zuweisen einer Struktur an eine andere Struktur mit demselben Typ (siehe nächstes Beispiel).
  - Rückgabe und Übergabe von Strukturen von einer und an eine Funktion.
  - Ermitteln der Adresse einer Struktur mit dem Adressoperator &.
  - Ermitteln der Größe einer Struktur mit dem `sizeof`-Operator.
- Für den Vergleich von Strukturen muss man aber eine **eigene** Funktion schreiben!

Beispiel (Strukturen)

```c
#include <stdio.h>
#include <stdlib.h>

struct person {
	char name[20];
	char adresse[30];
	int alter;
};

void output(struct person p) {
	printf("Personendaten:\n");
	printf("Name: %s\n", p.name);
	printf("Adresse: %s\n", p.adresse);
	printf("Alter: %d\n\n", p.alter);
}

int main(void) {
	struct person p1 = { "Alfred Zweistein", "Gasse 2, 1000 Wien", 25 };
	struct person p2 = { "Hansi Mozart", "Strasse 10, 1000 Wien" };
	struct person p3 = { .alter = 120, .name = "Guru" }; // C99!
	output(p1);
	output(p2);
	output(p3);
	p1 = p3;
	output(p1);
	return EXIT_SUCCESS;
}
```

Ausgabe:

```
Personendaten:
Name: Alfred Zweistein
Adresse: Gasse 2, 1000 Wien
Alter: 25

Personendaten:
Name: Hansi Mozart
Adresse: Strasse 10, 1000 Wien
Alter: 0

Personendaten:
Name: Guru
Adresse:
Alter: 120

Personendaten:
Name: Guru
Adresse:
Alter: 120
```

Größe von Strukturen

- Spezielle Anforderungen der Architektur an das Alignment (Ausrichtung der Daten an Wortgrenzen im Speicher) können den Speicherplatzbedarf beeinflussen.
- Beispiel (Windows-Rechner, 32-Bit):

```
struct person {
	char username[7];
	char password[8];
	int uid;
};
struct person2 {
	char username[8];
	char password[8];
	int uid;
};
struct person3 {
	char username[9];
	char password[8];
	int uid;
};
…
printf("%zu\n", sizeof(struct person));		// 20
printf("%zu\n", sizeof(struct person2));		// 20
printf("%zu\n", sizeof(struct person3));		// 24
```

Zeiger auf Strukturen

- Zeiger auf Strukturen lassen sich exakt so verwenden wie Zeiger auf normale Variablen.
- Beispiel: `struct person *p1;`
- Der Zugriff kann mit einer (umständlichen) Dereferenzierung erfolgen.

```
(*p1).name
```

- Die gängige Form ist aber ein Zugriff über „->“.

```
p1->name
```

- Beispiel (output-Funktion des Beispiels mit Zeiger auf Struktur)

```
void output(struct person *p) {
    printf("Personendaten:\n");
    printf("Name: %s\n", p->name);
    printf("Adresse: %s\n", p->adresse);
    printf("Alter: %d\n\n", p->alter);
}
```

Funktionen und Strukturen (Argumente)

- Variante 1
  - Einzelne Elemente einer Struktur einzeln übergeben
    - Einzelne Elemente einer Struktur werden wie bisherige Argumente behandelt.
- Variante 2
  - Struktur als Argument übergeben

```
void output(struct person p) { … }
```

  - Beispiel auf Folie 8
- Variante 3
  - Zeiger auf eine Struktur übergeben

```
void output(struct person *p) { … }
```

  - Beispiel auf Folie 10

Zeiger auf Struktur oder Struktur als Argument?

- Zeiger auf Struktur
  - Ist die schnellere Variante (wird nur Adresse übergeben).
  - Inhalt könnte verändert werden (nur Zeigerkopie)
    - `const` kann verwendet werden.
- Struktur als Argument
  - Struktur wird vollständig kopiert.
    - Einfach und sicher.
    - Aber langsamer und verbraucht mehr Platz (Speicher).

Funktionen und Strukturen (Rückgabe)

- Eine Funktion kann auch eine Struktur (oder auch einen Zeiger auf eine Struktur) zurückgeben.
- Beispiel für Struktur `person` aus dem vorherigen Beispiel

```
struct person create_person() {
    struct person p;
    // Strukturdaten verändern etc.
    …
    return p;
}
```

Arrays von Strukturen

- Man kann auch ein Array von Strukturen erstellen.
  - Wird oft für die Speicherung von „Listen“ zusammenhängender Daten verwendet
  - Beispiel: Array von Personendaten (in einer Struktur).

```
struct person {
        char name[20];
        char adresse[30];
        int alter;
};
…
struct person list[10] = {{ "Hansi Mozart", "Weg 10, 1000
Wien" }, { "Hubert Mozart", "Weg 11, 1000 Wien" }};
struct person p = { "Herbert Mozart", "Weg 12, 1000 Wien" };
list[2] = p;
printf("%s\n", list[0].name); // Hansi Mozart
printf("%s\n", list[1].name); // Hubert Mozart
printf("%s\n", list[2].name); // Herbert Mozart
```

Beispiel

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 5
#define MAXCHAR 30

struct music_tag {
	char titel[MAXCHAR];
	char kuenstler[MAXCHAR];
	short jahr;
};

void output(struct music_tag *song) {
	printf("\n\n");
	printf("Titel     :  %s", song->titel);
	printf("Kuenstler :  %s", song->kuenstler);
	printf("Jahr      :  %hd\n", song->jahr);
}

int main(void) {
	struct music_tag data[MAX];

	for (int i = 0; i < MAX; i++) {
		printf("Titel     :  ");
		fgets(data[i].titel, MAXCHAR, stdin);
		printf("Kuenstler :  ");
		fgets(data[i].kuenstler, MAXCHAR, stdin);
		printf("Jahr      :  ");
		do {
			scanf("%hd", &data[i].jahr);
		} while (getchar() != '\n');
	}
	for (int i = 0; i < MAX; i++) {
		output(&data[i]);
	}
	return EXIT_SUCCESS;
}
```

Für das einfachere Einlesen von Strings wird hier `fgets` verwendet!
Erklärung folgt noch! Manual-Seite lesen!

Zusammengesetztes Literal

- Seit C99 gibt es auch zusammengesetzte Literale für Strukturen.
- Form (für Beispiel von vorherigen Folien)
  - `(struct person){ "Hans Huber", "Weg 10, 1000 Wien" } …`
- Verwendung bei Funktionsaufrufen

```
output((struct person){ "Hans Huber", "Weg 10, 1000 Wien" });
```

- Verwendung bei Zuweisungen

```
struct person p;
…
p = (struct person){ "Hans Huber", "Weg 10, 1000 Wien" };
```

oder

```
struct person list[10];
…
list[0] = (struct person){ "Hans Huber", "Weg 10, 1000 Wien" };
```

Strukturen in Strukturen (1)

- Strukturen können auch in Strukturen verwendet werden - Beispiel:

```
struct details {
      short bitrate;
      short spielzeit;
      int abtastrate;
};

struct music_tag {
      char titel[30];
      char kuenstler[30];
      short jahr;
};

struct meta_info {
      struct music_tag music;
      struct details info;
};
```

Strukturen in Strukturen (2)

- Gesamtgröße ergibt sich aus der Addition der einzelnen Komponenten.
- Eine Struktur darf nicht die eigene Struktur wieder enthalten.
  - Ein Zeiger auf die eigene Struktur ist aber schon erlaubt, und das wird noch ausführlich besprochen!
- Beispiele für Zugriff (Beispiel von vorheriger Folie):
  - `m.info.abtastrate = 100;`
  - `printf("%d",m.info.abtastrate);`

Zeiger und Strukturen

- Zeiger in Strukturen
  - Zeiger können auch in Strukturen verwendet werden.
  - Dabei sind die Elemente Zeiger, d.h. sie repräsentieren wiederum eine Anfangsadresse.
  - Diese Elemente sollten auf Speicherplatz zeigen, der dynamisch angelegt wurde!
- Dynamische Strukturarrays oder Arrays von Strukturzeigern sind auch möglich.

Beispiel

- Das externe Beispiel `matrix2.c` fasst einige Aspekte von Strukturen in einem Programm zusammen
  - Verwendung einer Struktur für gemeinsame Daten
    - Matrix (Zeiger in Struktur)
    - Dimension 1
    - Dimension 2
  - Verwendung von Strukturen als Übergabeparameter
  - Verwendung von Strukturen bei der Rückgabe
  - Dynamische Allokation bei Strukturen (Matrix)

typedef (1)

- Vereinbarung von neuen Typnamen
  - Vorsicht, keine Textersetzung (wie bei `#define`).
  - Compiler kennt neuen Datentyp.
- Beispiele

```
typedef unsigned long un64;
typedef struct treenode tn;
```

- Sinn: Zusätzlichen Namen für einen existenten Typ einführen.
  - Gleiche Semantik, gleiche Operationen etc.!
- Kann bei Vereinbarungen, Umwandlungen usw. verwendet werden.

```
un64 l1, maxlen;
```

- Einsatz
  - Portabilität
  - Verständlichkeit (das ist aber eine Streitfrage!)

typedef (2)

- Beispiel

```
typedef struct person {
    char username[8];
    char password[8];
    int uid;
} user;
user user1 = {"Hans", "magic", 2 };
```

- Aber:
  - typedef sollte nur sparsam eingesetzt werden!

offsetof-Makro

- Makro definiert in `<stddef.h>`
- Damit kann man für ein bestimmtes Strukturelement den Abstand in Bytes von der Anfangsadresse ermitteln.
- Beispiel

```
typedef struct zeit {
  unsigned int stunde;
  unsigned int minute;
  unsigned int tag;
  unsigned int monat;
  unsigned int jahr;
  unsigned int sommerzeit;
} Zeit_t;
…
```

  - `printf("%d\n",offsetof(Zeit_t,jahr)); // 16 (Windows)`

UNIONS

Unions

- Zwischen Unions und Strukturen bestehen außer einem anderen Schlüsselwort keine syntaktischen Unterschiede.
- Der Hauptunterschied liegt in der Art und Weise, wie mit dem Speicherplatz der Daten umgegangen wird.
- Eine Union ist eine Variable, die (zu verschiedenen Zeitpunkten) Objekte mit verschiedenen Datentypen und Größen enthält.

Definition und Sinn

- Beispiel

```
union u_tag {
    int ival;
    float fval;
    char *sval;
} u;
```

- Enthält nur **einen** Wert
  - Jedes Element in der Union hat **dieselbe** Anfangsadresse.
  - **u** ist groß genug, um den größten der drei Datentypen (in diesem Fall `float`) aufzunehmen (Alignment beachten!).
  - Kann nur mit einem Wert initialisiert werden, der zum Typ der ersten Alternative passt.
  - Wird einer der Werte gesetzt, dann werden andere schon existierende Werte überschrieben.
- Zugriff mit „.“: `u.ival`

Beispiel

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef union variant {
      char cval[20];
      short sval;
      int ival;
      double dval;
} variant_t;

int main(void) {
      variant_t var;
      printf("Union = %d Bytes\n", sizeof(var));
      var.dval = 5.5f;
      printf("%f\n", var.dval);
      var.ival = 12;
      printf("%d\n", var.ival);
      strcpy(var.cval, "HALLO");
      printf("%s\n", var.cval);
      return EXIT_SUCCESS;
}
```

Ausgabe:

```
Union = 24 Bytes
5.500000
12
HALLO
```

Nutzen von Unions

- Mit Unions kann man Speicherplatz einsparen.
  - Ein gemeinsamer Bezeichner.
  - Jedes Element der Union macht nur einzeln einen Sinn.
- Initialisierung
  - Es wird immer nur das erste Element einer Union initialisiert.
  - In C99 kann man bei der Initialisierung Elementbezeichner angeben (wie bei Strukturen).

BITFELDER

Bitfelder

- Als Bitfelder können einzelne Elemente einer Struktur oder einer Union deklariert werden.
  - Diese Elemente bestehen aus einer ganzzahligen Variable, die wiederum aus einer bestimmten Anzahl von Bits besteht.
  - Man kann Bit für Bit innerhalb eines Bytes arbeiten.
  - Auf Bitfelder kann wie auf gewöhnliche Strukturelemente zugegriffen werden.
- Anwendungsbereiche für Bitfelder
  - Einsparung von Speicherplatz (z.B. bei Eingebetteten Systemen)
  - Zugriff auf die Hardware bzw. Peripherie einer Controllers
    - Informationen sind hier meist bitweise in Registern kodiert.

Deklaration

- Syntax
  - `Typ Elementname : Breite`
  - Typ muss ein ganzzahliger Typ sein.
  - Mit dem Bezeichner Elementname greift man auf das Element zu.
  - Breite gibt die Anzahl der Bits an.
- Beispiel (normale Struktur, Struktur mit Bitfeldern)

```
typedef struct zeit {
  unsigned int stunde;
  unsigned int minute;
  unsigned int tag;
  unsigned int monat;
  unsigned int jahr;
  unsigned int sommerzeit;
} Zeit_t;
```

```
typedef struct zeit {
  unsigned int stunde:5;
  unsigned int minute:6;
  unsigned int tag:5;
  unsigned int monat:4;
  unsigned int jahr:11;
  unsigned int sommerzeit:1;
} Zeit_t;
```

Beispiel

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct zeit {
	unsigned int stunde:5;
	unsigned int minute:6;
	unsigned int tag:5;
	unsigned int monat:4;
	unsigned int jahr:11;
	unsigned int sommerzeit:1;
} zeit_t;

void zeit(const zeit_t *z) { printf("%02u:%02u Uhr\n", z->stunde, z->minute); }
void datum(const zeit_t *d) { printf("%02u.%02u.%04u\n", d->tag, d->monat, d->jahr); }

int main(void) {
	zeit_t z1;
	zeit_t z2 = { 17, 22, 17, 6, 2010, 1 };
	z1.stunde = 23;
	z1.minute = 55;
	z1.tag = 18;
	z1.monat = 6;
	z1.jahr = 2010;
	z1.sommerzeit = 1;
	zeit(&z1);
	datum(&z2);
	return EXIT_SUCCESS;
}
```

Ausgabe:

```
23:55 Uhr
17.06.2010
```

Einschränkungen

- Bitfelder können keine adressierbaren Speicherstellen belegen.
  - Adressoperator kann nicht auf einzelne Bitfelderelemente angewendet werden.
  - Bitfeld kann nicht mit `scanf` eingelesen werden.
  - Es kann kein Array von Bitfelderelementen verwendet werden.
  - Es kann nicht das `offsetof`-Makro auf Bitfelderelemente angewendet werden.
- Die Anordnung der Bits eines Bitfeldelementes im Speicher ist nicht standardisiert.
  - Dies kann von Compiler zu Compiler unterschiedlich sein (Architektur!).
- Der Zugriff auf Bitfeldelemente ist langsamer als der auf herkömmliche Datentypen!
  - Elemente haben keine einheitliche Größe.