universität innsbruck

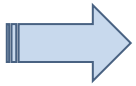
Strukturen, Unions, Bitfelder

Einführung in die Programmierung

Michael Tschuggnall
Universität Innsbruck

Überblick

- Einführung
- Vorschau
- Variablen und Datentypen
- Operatoren und Ausdrücke
- Kontrollstrukturen
- Funktionen
- Arrays
- Zeiger
- Strukturen, Unions, Bitfelder
- Speicherklassen
- Dynamische Speicherverwaltung
- Präprozessor
- Modulare Programmierung
- Ein- und Ausgabe
- gcc und make



- Strukturen
- Unions
- Bitfelder

Strukturen

Strukturen

- Eine Struktur ist ein selbst definierter zusammengesetzter Datentyp.
- Sie besteht aus Komponenten (Members), die einen Typ und einen Namen haben.
  - Diese Komponenten können **unterschiedliche Typen** haben.
- Die Deklaration bzw. Definition beginnt mit dem Schlüsselwort `struct` und enthält eine Liste von Komponenten in geschweiften Klammern.
  - Die Namen der Komponenten einer Struktur müssen eindeutig sein.
  - Eine Struktur muss mindestens eine Komponente haben.
- Allgemeine Form:

```
struct [tag_name] { members };
```

Beispiel Punkt in 2D (1)

- Deklaration eines Punkts:

```
1 struct point {
2     int x;
3     int y;
4 };
```

  - Der Bezeichner `point` wird als Tag des Struktur-Typs bezeichnet.
  - Die Bezeichner `x` und `y` sind die Namen der Komponenten.
- Im Gültigkeitsbereich des Struktur-Typs können Variablen mit diesem Typ deklariert werden:

```
1 struct point my_point;
```

Beispiel Punkt in 2D (2)

- Die Deklaration und Definition einer Struktur kann auch zusammengefasst werden:

```
1 struct point {
2     int x;
3     int y;
4 } my_point;
```

  - Neben dem Deklarieren des Struktur-Typs mit dem Tag `point` wird eine Variable `my_point` vom Typ dieser Struktur definiert.
- Das Tag des Struktur-Typs ist optional.

```
1 struct {
2     int x;
3     int y;
4 } my_point;
```

Operatoren bei Struktur-Typen

- Zuweisungsoperator (=)
- Adressoperator (&)
- Punktoperator (`.`) und Pfeiloperator (`->`)
  - Zugriff auf eine Komponente einer Struktur.
- sizeof-Operator
  - Ermitteln der Größe einer Struktur in Bytes.
- _Alignof-Operator
  - Ermitteln des Alignments einer Struktur in Bytes.

Initialisierung bei der Definition

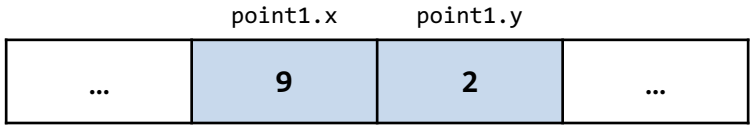
- Die Komponenten einer Strukturvariable können bei der Definition mit einer nicht leeren, kommaseparierten Liste, welche von geschwungenen Klammern umschlossen ist, initialisiert werden.
  - Die Komponenten können in der Reihe der Deklaration initialisiert werden.
  - Alternativ können die Komponenten über den Namen angesprochen werden.
- In der Initialisierungsliste müssen nicht alle Komponenten initialisiert werden.
  - Alle Komponenten ohne Wert in der Initialisierungsliste werden mit dem Null-Wert des Datentyps initialisiert.

Beispiel Initialisierung

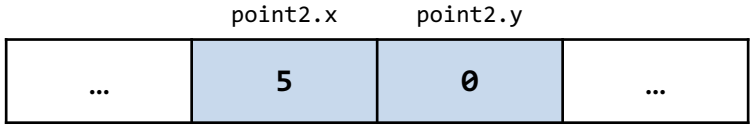
```
1  struct point {
2      int x;
3      int y;
4  };
```

- Initialisierung in der Reihe der Deklaration:

```
struct point point1 = {9, 2};
```

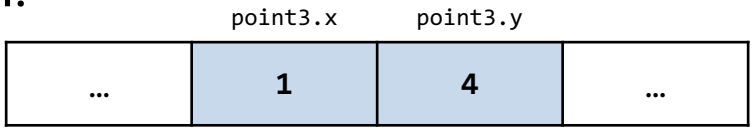



```
struct point point2 = {5};
```

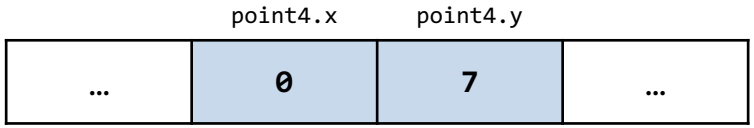


- Initialisierung mit Komponentennamen:

```
struct point point3 = {.x = 1, .y = 4};
```




```
struct point point4 = {.y = 7};
```



Zugriff auf Komponenten

- Der Punktoperator (`.`) und der Pfeiloperator (`->`) können für den Zugriff auf Komponenten einer Struktur verwendet werden.
- Beide Operatoren sind binäre Operatoren.
  - Der linke Operand ist beim Punktoperator ein Struktur-Typ.
  - Der linke Operand ist beim Pfeiloperator ein Zeiger auf einen Struktur-Typ.
  - Der rechte Operand ist der Bezeichner einer Komponente des Struktur-Typs.

Beispiel Zugriff auf Komponenten

```c
struct point {
    int x;
    int y;
};
struct point my_point = {9, 2};
struct point *ptr_my_point = &my_point;
```

- Zugriff auf die Komponente `x` von `my_point`:

  `my_point.x`

- Zugriff auf die Komponente y von `ptr_my_point` mit Punktoperator:

  `(*ptr_my_point).y`

  - Der Punktoperator hat eine höhere Priorität als der Dereferenzierungsoperator.
  - Die runden Klammern sind notwendig.

- Zugriff auf die Komponente y von `ptr_my_point` mit Pfeiloperator:

  `ptr_my_point->y`

  - Bei Zeigern auf Strukturen ist für den Zugriff auf Komponenten der Pfeiloperator zu bevorzugen.

Größe von Strukturen

- Die minimale Größe einer Struktur ergibt sich aus der Addition der Größen der einzelnen Komponenten.
- Spezielle Anforderungen der Architektur an das Alignment (Ausrichtung der Daten an Wortgrenzen im Speicher) können den Speicherplatzbedarf beeinflussen.
  - Der Compiler kann für das Alignment Padding zwischen Komponenten und am Ende der Struktur einführen.
- Die Ordnung der Komponenten kann die Größe beeinflussen.
- Beispiel (Linux-Rechner, 64-Bit):

```
1  struct my_struct1 { // 24 bytes
2      int *p;  // 8 bytes
3      char c1; // 1 byte
4               // 3 bytes padding
5      int i;   // 4 bytes
6      char c2; // 1 byte
7               // 7 bytes padding
8  };
```

```
1  struct my_struct2 { // 16 bytes
2      int *p;  // 8 bytes
3      char c1; // 1 byte
4      char c2; // 1 byte
5               // 2 bytes padding
6      int i;   // 4 bytes
7  };
8
```

Typen von Komponenten

- Funktionen können nicht in Strukturen definiert bzw. deklariert werden.
- Arrays ohne Längenangabe dürfen nur bei Strukturen mit mehr als einer Komponente als letzte Komponente verwendet werden.
  - Ein solches Array wird als *flexible array member* bezeichnet.
  - Der C-Standard erlaubt für solche Arrays keine Initialisierung über die Initialisierungsliste.
  - Der Speicher muss über die dynamische Speicherverwaltung reserviert werden.
- Arithmetische Typen und alle anderen zusammengesetzten Typen können für die Typen von Komponenten verwendet werden.
  - Der eigene Struktur-Typ darf nicht als Komponententyp verwendet werden.
    - Ein Zeiger auf die eigene Struktur ist allerdings möglich!
  - Bei Arrays muss eine fixe Größe angegeben werden.

Beispiel Strukturen in Strukturen

```c
struct address {
    int postal_code;
    char city[32];
    char street[32];
    char country[64];
};

struct person {
    char name[64];
    int year_of_birth;
    struct address address;
};
```

```c
struct person mozart = {"Wolfgang Amadeus Mozart", 1756,
                        {5020, "Salzburg", "Getreidegasse 9", "Austria"}};
```

Struktur als Parameter bzw. Argument

- Variante 1: Elemente einer Struktur einzeln übergeben.

```
void print_address(int postal_code, char *city, char *street,
                   char *country);
```

  - Alle Argumente werden kopiert.
  - Für jede Komponente wird ein Parameter benötigt.

- Variante 2: Struktur als Argument übergeben.

```
void print_address(struct address address);
```

  - Struktur wird vollständig kopiert.

- Variante 3: Zeiger auf eine Struktur übergeben.

```
void print_address(struct address *address);
```

  - Nur die Adresse der Struktur wird übergeben.
  - Inhalt könnte verändert werden.
    - `const` kann verwendet werden, um Änderungen zu verhindern.

Beispiel Struktur übergeben

example_print_person1.c

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

struct address {
    int postal_code;
    char city[32];
    char street[32];
    char country[64];
};

struct person {
    char name[64];
    int year_of_birth;
    struct address address;
};

void print_address(struct address address) {
    printf("Address: %s, %d %s\n", address.street, address.postal_code, address.city);
    printf("Country: %s\n", address.country);
}

void print_person(struct person p) {
    printf("Name: %s\n", p.name);
    printf("Year of birth: %d\n", p.year_of_birth);
    print_address(p.address);
}
```

Beispiel Zeiger auf eine Struktur übergeben

example_print_person2.c

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

struct address {
    int postal_code;
    char city[32];
    char street[32];
    char country[64];
};

struct person {
    char name[64];
    int year_of_birth;
    struct address address;
};

void print_address(const struct address *address) {
    printf("Address: %s, %d %s\n", address->street, address->postal_code, address->city);
    printf("Country: %s\n", address->country);
}

void print_person(const struct person *p) {
    printf("Name: %s\n", p->name);
    printf("Year of birth: %d\n", p->year_of_birth);
    print_address(&p->address);
}
```

Struktur als Rückgabetyp einer Funktion

- Eine Funktion kann auch eine Struktur (oder auch einen Zeiger auf eine Struktur) zurückgeben.
- Beispiel:

```
1  struct address {
2      int postal_code;
3      char city[32];
4      char street[32];
5      char country[64];
6  };
7
8  struct person {
9      char name[64];
10     int year_of_birth;
11     struct address address;
12 };
13
14 struct person create_person() {
15     struct person p;
16     …
17     return p;
18 }
```

Aufgabe Strukturen und Funktionen

- Gegeben seien die folgenden Deklarationen.

```c
struct lecture_hall {
    char name[64];
    unsigned number_of_seats;
};
void function1(char *name, unsigned number_of_seats);
void function2(struct lecture_hall lecture_hall);
void function3(struct lecture_hall *lecture_hall);
```

- Geben Sie für jeden Funktionsaufruf an, welche Komponenten der Struktur `hs_a` durch den jeweiligen Funktionsaufruf verändert werden können!

```c
struct lecture_hall hs_a = {"HS A", 305};
function1(hs_a.name, hs_a.number_of_seats);
function2(hs_a);
function3(&hs_a);
```

Aufgabe Strukturen und Funktionen – Lösung

- Gegeben seien die folgenden Deklarationen.

```c
struct lecture_hall {
    char name[64];
    unsigned number_of_seats;
};
void function1(char *name, unsigned number_of_seats);
void function2(struct lecture_hall lecture_hall);
void function3(struct lecture_hall *lecture_hall);
```

- Geben Sie für jeden Funktionsaufruf an, welche Komponenten der Struktur `hs_a` durch den jeweiligen Funktionsaufruf verändert werden können!

```c
struct lecture_hall hs_a = {"HS A", 305};
function1(hs_a.name, hs_a.number_of_seats); // name
function2(hs_a);                            // neither
function3(&hs_a);                           // both
```

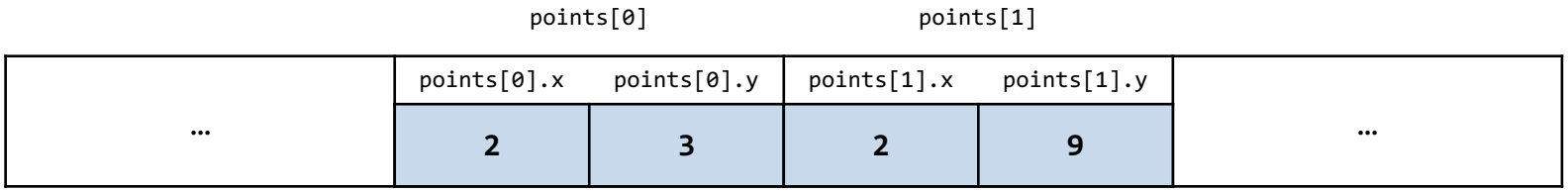
Arrays von Strukturen

- Struktur-Typen können als Elementtyp bei Arrays verwendet werden.
- Beispiel:

```c
struct point {
    int x;
    int y;
};

struct point points[] = {{2, 3}, {2, 9}};
```

- Visualisierung des Arrays `points`:



Zusammengesetzte Literale (Compound Literals)

- Seit C99 können Literale verwendet werden, die eine Struktur darstellen.
- Form (für Beispiel der vorherigen Folie):

```
(struct point){3, 2}
```

- Verwendung bei Zuweisungen:

```
struct point p;
…
p = (struct point){3, 2};
```

typedef (1)

- Es wird ein zusätzlicher Name für einen existierenden Typ eingeführt.
  - Gleiche Semantik, gleiche Operationen etc.!
  - Vorsicht, keine Textersetzung (wie bei `#define`).
  - Compiler kennt den Aliasnamen.
    - Der Alias-Typ ist kompatibel mit dem bestehenden Typ.
- Beispiele:

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

- Kann bei Deklarationen, Casts usw. verwendet werden.

```
un64 max_len;
tn node;
```

- Einsatz:
  - Alias für komplexe Typen
  - Portabilität
  - Verständlichkeit (das ist aber eine Streitfrage!)

typedef (2)

- Beispiel:

```c
typedef struct point {
    int x;
    int y;
} point_t;

point_t my_point = {1, 5};
```

- `typedef` sollte nur sparsam eingesetzt werden!

offsetof-Makro

- Durch das Makro `offsetof` kann für eine Komponente einer Struktur der Abstand in Bytes von der Anfangsadresse ermittelt werden.
- Das Makro ist in der Header-Datei `stddef.h` definiert.
- Das Ergebnis hat den Typ `size_t`.
- Allgemeine Form:

```
offsetof(Struktur_Typ, Komponente)
```

Beispiel offsetof-Makro

example_offsetof.c

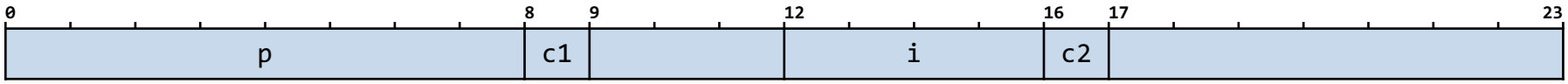
```c
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

struct my_struct1 {
    int *p;
    char c1;
    int i;
    char c2;
};

int main(void) {
    printf("%zu", offsetof(struct my_struct1, c2));
    return EXIT_SUCCESS;
}
```

Ausgabe auf der Standardausgabe (plattformabhängig):

16



Unions

Unions (1)

- Zwischen Unions und Strukturen bestehen außer einem anderen Schlüsselwort keine syntaktischen Unterschiede.
- Im Gegensatz zu Strukturen werden bei Unions die Komponenten aber nicht hintereinander im Speicher abgebildet, sondern alle Komponenten beginnen an derselben Adresse.
  - Eine Unionvariable ist eine Variable, die (zu verschiedenen Zeitpunkten) Objekte mit verschiedenen Datentypen und Größen enthält.
- Die Union ist so groß wie die Größe der größten Komponente (inklusive Padding).

Unions (2)

- Beispiel:

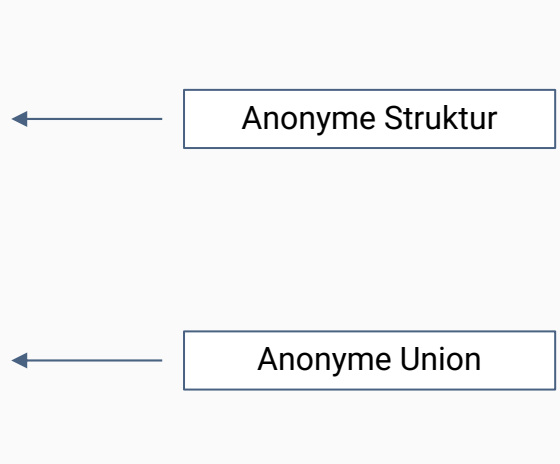
```
1  union u_tag {
2      int i;
3      float f;
4      char *s;
5  } u;
```

- Enthält nur **einen** Wert:
  - Jedes Element in der Union hat **dieselbe** Anfangsadresse.
  - **u** ist groß genug, um den größten der drei Datentypen inklusive Padding aufzunehmen (Alignment beachten!).
  - Für das Initialisieren kann eine Initialisierungsliste mit einem Element verwendet werden.
    - Wird kein Element spezifiziert, wird die erste Alternative initialisiert.
  - Wird einer der Werte gesetzt, dann werden andere schon existierende Werte überschrieben.

Anonyme Strukturen und Unions

- Strukturen oder Unions, welche als Komponenten in Strukturen bzw. Unions verwendet werden, können ohne Komponentennamen deklariert werden.
  - Diese namenlosen Komponenten werden als anonyme Strukturen bzw. Unions bezeichnet.
  - Der Zugriff erfolgt, als wären die Komponenten Teil der äußeren Einheit.
- Beispiel:

```
1  struct my_struct {
2      char member_1;
3      struct {
4          int member_2;
5          float member_3;
6      };
7      union {
8          unsigned member_4;
9          double member_5;
10     };
11 };
```



Beispiel

example_union.c

```c
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#define PI 3.14159265358979323846264338327950288

enum shape_type {CIRCLE, RECTANGLE};

struct shape {
    enum shape_type type;
    union {
        double radius;
        struct {
            double length;
            double width;
        };
    };
};

double area(const struct shape *s) {
    switch (s->type) {
        case CIRCLE:
            return s->radius * s->radius * PI;
        case RECTANGLE:
            return s->width * s->length;
    }
    return -1;
}

int main(void) {
    struct shape my_shapes[] = {{CIRCLE, {.radius = 2}}, {RECTANGLE, {.length = 4, .width = 2}}};
    for (size_t i = 0; i < sizeof(my_shapes)/ sizeof(*my_shapes); ++i) {
        printf("%f\n", area(my_shapes + i));
    }
    return EXIT_SUCCESS;
}
```

Ausgabe auf der Standardausgabe:

```
12.566371
8.000000
```

Bitfelder

Bitfelder

- Ein Bitfeld ist eine Komponente einer Struktur oder Union mit einer bestimmten Anzahl an Bits.
  - Die Anzahl der Bits eines Bitfelds wird bei der Deklaration bestimmt.
  - Werden mehrere Bitfelder hintereinander deklariert, können sie in eine Speichereinheit verpackt werden.
  - Diese Komponenten sind vom Typ `_Bool`, `signed int`, `unsigned int` oder einem implementierungsabhängigen Typ.
    - Bitfelder mit `unsigned` werden als vorzeichenlose Zahlen interpretiert.
    - Bitfelder mit `signed` werden als vorzeichenbehaftete Zahlen interpretiert.
    - Bitfelder mit `int` werden abhängig von der Implementierung als vorzeichenlose oder vorzeichenbehaftete Zahlen interpretiert.
      - Achtung: Bei Bitfeldern ist `int` kein Synonym für `signed int`!
  - Auf Bitfelder kann wie auf gewöhnliche Komponenten zugegriffen werden.
- Anwendungsbereiche für Bitfelder:
  - Einsparung von Speicherplatz (z.B. bei eingebetteten Systemen).
  - Zugriff auf die Hardware bzw. Peripherie eines Controllers.
    - Informationen sind hier meist bitweise in Registern kodiert.

Deklaration

- Syntax:

  `Typ Elementname : Breite`

  - Breite gibt die Anzahl der Bits an.
    - Die Breite ist ein konstanter ganzzahliger Ausdruck mit einem Wert größer gleich 0.
    - Hat die Breite den Wert 0, darf kein Elementname angegeben werden. Das nächste Bitfeld beginnt bei einer neuen Speichereinheit.

- Beispiele für Bitfelder (Zweierkomplement):

Bitfeld	Wertebereich
`unsigned int a : 3;`	`0, …, 7`
`signed int b : 3;`	`-4, …, 3`
`int c : 3;`	`0, …, 7 oder -4, …, 3`

- Beispiel:

```
1  struct date {
2      unsigned int day : 5;
3      unsigned int month : 4;
4      unsigned int year : 12;
5  };
```

Beispiel Bitfelder

example_bitfield.c

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

struct date {
    unsigned int day : 5;
    unsigned int month : 4;
    unsigned int year : 12;
};

void print_date(const struct date *d) {
    printf("%02u.%02u.%04u\n", d->day, d->month, d->year);
}

int main(void) {
    struct date d = {1, 12, 2020};
    print_date(&d);
    return EXIT_SUCCESS;
}
```

Ausgabe auf der Standardausgabe:

```
01.12.2020
```

Einschränkungen

- Bitfelder können Speicherstellen belegen, die nicht adressierbar sind.
  - Der Adressoperator kann nicht auf Bitfelder angewendet werden.
  - Das `offsetof`-Makro kann nicht auf Bitfelder angewendet werden.
- Der `sizeof`-Operator kann nicht auf Bitfelder angewendet werden.
- Es kann kein Array von Bitfeldern verwendet werden.
- Die Reihenfolge der Anordnung von Bitfeldern, die in eine Speichereinheit verpackt werden, ist implementierungsabhängig.
- Der Zugriff auf ein Bitfeld ist langsamer als der auf herkömmliche Datentypen!
  - Bitfelder haben keine einheitliche Größe.
- Nur die Typen `_Bool`, `int` und `unsigned` dürfen garantiert als Typ eines Bitfelds verwendet werden.
  - Die Unterstützung weiterer Typen ist implementierungsabhängig.