Kapitel 08 - Eigene Datenstrukturen

Peter Ulbrich

AG Systemsoftware

Veranstaltungswebseite

Einleitung

- Bisher: **Elementare** Datentypen wie `int`, `float`, ...
- Problem: Viele Informationen sind Kombinationen von mehreren Variablen
  - Beispiel Koordinate: Ein Paar aus zwei ganzen Zahlen
  - Beispiel Student: Matrikelnummer, Vorname, Nachname, ...
- Naiver Ansatz: Einfach zwei Variablen deklarieren/initialisieren
  - Für eine Instanz trivial: `int x, int y;`
  - Was aber, wenn zwei, drei, ..., N Koordinaten benötigt werden?
- In Kapitel 2 kurz vorgestellt: Schlüsselwort `struct`

```
struct Koordinate {
    int x;
    int y;
};
```

Eigene Datentypen

- Selbstdefinierte Datenstrukturen (*structs*) werden aus anderen Datentypen zusammengesetzt
- Deklaration umfasst Namen und Komponenten
  - Einzelne Komponenten heißen **Member**
  - Keine Speicherbelegung bei der Deklaration!

  → Geschieht erst später

```
struct Adresse {
    unsigned int plz;
    char ort[30];
    char strasse[30];
};
int main() {
    struct Adresse addr = { 44227,
        "Dortmund", "OH 14" };
    return 0;
}
```

- Schlüsselwort `struct` kennzeichnet eigene Datentypen, gefolgt von Typbezeichner

Aufbau von Strukturen

- Member-Deklaration innerhalb eines Codeblocks `{}`
- Deklaration wird per Semikolon terminiert

  für Member und Datenstruktur
- Zugriff auf einzelnen Member mit Punktoperator(`.`)
  - Gilt für Lesen und Schreiben gleichermaßen

```c
#include <stdio.h>
struct Adresse {
    unsigned int plz;
    char ort[30];
    char strasse[30];
};
int main() {
    struct Adresse addr = { 44227,
        "Dortmund", "OH 14" };
    addr.plz = 4711;
    printf("PLZ: %d", addr.plz);
    return 0;
}
```

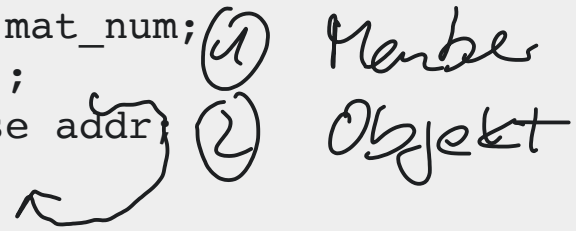
Verschachelte Datenstrukturen

- Strukturen dürfen wieder Strukturen enthalten
- **Aber:** Keine zirkulären Abhängigkeiten
  - Ausnahme: Zeiger auf Datentyp (→ nächstes Kapitel)
  - Gilt auch für kreisförmige Abhängigkeiten

```c
struct Foo { struct Foo f; }; // Error
```

```c
struct Foo { struct Bar b; };
struct Bar { struct Foo f; }; // Error
```

```c
struct Adresse {
    unsigned int plz;
    char ort[30];
    char strasse[30];
};
struct Student {
    unsigned int mat_num;
    char name[30];
    struct Adresse addr;
};
int main() {
    struct Adresse addr = { 44227,
        "Dortmund", "OH 14" };
    struct Student stud = { 254711,
        "Musterstudent", addr };
    return 0;
}
```



Initialisierung von Datenstrukturen

- Erst bei der Initialisierung wird Speicher angelegt
- Daher: Rekursive Definition einer Struktur verboten
  - Unbeschränkte Rekursion bei der Initialisierung
  - Größe nicht bestimmbar
- Initialisierung über *Initializer List*:
  - Auflistung der Memberwerte in Reihenfolge der Deklaration
  - Kommasepariert, von Klammern ({}) umfasst (wie Array-Initialisierung)
  - `struct Adresse addr = { 44227, "Dortmund", "Otto-Hahn-Straße 14" };`

Unterschiedliche Art der Initialisierung

- Keine Initialisierung

```
struct Adresse addr;
```

- **Explizite** *Initializer List*
  - Initialisierung in Reihenfolge der Deklaration
  - Der Speicher nicht genannter Member wird mit `0` initialisiert

```
struct Adresse addr = { 44227, "Dortmund", "Otto-Hahn-Straße 14" };
struct Adresse addr = { 44227, "Dortmund", }; // Ok (strasse mit 30 Null-Bytes)
```

- **Leere** *Initializer List*
  - Initialisiert alle Werte des Structs zu `0`
  - In C++ in allen Versionen möglich, in C erst seit C23

```
Adresse addr = {};
```

Designated Initializer List

- Initialisierung einzelner Member durch explizite Namensnennung (*Designation*)
- Initialisierung in **beliebiger** Reihenfolge
- Möglich ab C99 bzw. C++20
- Nicht genannte Member werden ebenfalls mit `0` initialisiert

C Run ▶

```c
#include <stdio.h>
struct Adresse {
    unsigned int plz;
    char ort[30];
    char strasse[30];
};

int main() {
    struct Adresse addr = {
        .ort = "Dortmund",
        .strasse = "OH 14", };
    printf("Adresse: %d %s, %s\n",
           addr.plz, addr.ort,
           addr.strasse);
    return 0;
}
```

Einschub: `typedef`

- Verwendung von Schlüsselwort `struct` in C Pflicht
- Realisierung mittels Schlüsselwort `typedef`

```
typedef struct { int x; int y; } Koordinate;
Koordinate k { 1, 3};
```

Einschub: `typedef`

- Führt einen neuen Namen (Alias) für bestehenden Datentyp ein
- Häufiger Anwendungszweck: Vereinfachen langer oder komplizierter Typen
- Funktionsweise ähnlich wie `#define`, aber **nicht** Teil des Präprozessors

  → Teil des Compilers (und auf Typen beschränkt)
- Beispiel:

```
typedef __INT64_TYPE__ int64_t;
typedef __UINT8_TYPE__ uint8_t;
```

(Auszug aus den Quelldateien des Clang-Compilers)

Einschub: `typedef`

- Einsatz bei Datenstruktur auf zwei Weisen

1. Aliases für existierende, benannte Datenstruktur (*Tagged Struct*)
2. Definierung eines anonymen Datenstruktur (*Anonymous Struct*)

C Run ▶

```c
#include <stdio.h>
#include <stdint.h>
struct Int{ uint32_t value; };
typedef struct Int Int;
typedef struct { uint32_t value; } Int2;

int main () {
    // Kein 'struct' nötig
    Int i = { 3 };
    Int2 i2 = { 2 };
    printf("%d, %d\n", i.value, i2.value);
    return 0;
}
```

Gleichzeitige Deklaration und Initialisierung

- Auch möglich: Gleichzeitige Deklaration und Initialisierung möglich
- Zweck: Struktur mit lokaler Beschränkung
  - Wird nur im aktuellen Kontext verwendet
- Nachteil: **Deklaration** ist **nicht allgemein** verwendbar
  - Nur die aktuelle Variable hat die deklarierte Struktur

C Run ▶

```c
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>

int main() {
    struct { // Anonyme Deklaration
        bool enabled;
        uint8_t value;
    } cfg = { true, 33 }; // Init
    printf("Intensity: %d\n", cfg.value);
    return 0;
}
```

→ Für häufiger eingesetzte Strukturen lohnt sich separate Deklaration als *Tagged Struct*

Unions

- Ene weitere Datenstruktur: `union`
- Prinzipiell sehr ähnlich zu `struct`
- Unterschied: Member teilen sich den Speicher
- Speicherplatz ergibt sich aus größtem Element
- Adressierung der Member ist frei möglich

C Run ▶

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

union IntChar {
    uint32_t value;
    uint8_t arr[4];
};

int main() {
    union IntChar u1 = { 65 }; // 'A'
    printf("%c\n", u1.arr[0]);
    printf("%d\n", u1.value);
    u1.arr[3] = 55;
    printf("%c\n", u1.arr[0]);
    printf("%d\n", u1.value);
    return 0;
}
```

Unions

- Besonderheit: Explizite Initialisierung gilt nur für das erste Element
- Für alle anderen gilt: Initialisierung mit designierter Variable

C Run ▶

```c
#include <stdint.h>
union IntChar {
    uint32_t value;
    uint8_t arr[4];
};

int main() {
    IntChar u1 = { 15 }; // Ok
    IntChar u2 = { .arr = {'a','b','c','d'}}; // Ok
    // IntChar u3 = { {'a','b','c','d'} }; // Error
    return 0;
}
```

Bitfelder

- Bisher: Datenstrukturelemente belegen ganze Bytes
- Jetzt: Einschränkung auf einzelne Bits
- Sogenannte Bitfelder *(Bit-fields)* belegen nur N Bits
- Notation ist simpel: Bitbreite folgt auf Elementdeklaration

  → *Typ Bezeichner*: *Breite*

```c
#include <stdint.h>
struct Byte {
    uint8_t lower_half: 4;
    uint8_t upper_half: 4;
};

int main() {
    struct Byte b { 0x7, 0xF};
    b.lower_half = 1;
}
```

Bitfelder

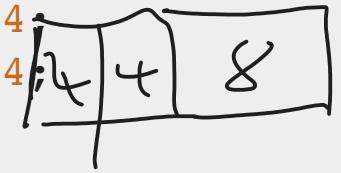
- Achtung: Wertebereich ist auf Bitbreite beschränkt
- Achtung: Compiler **darf** Bitfeld im Speicher selbst verteilen!
- Lösung: Im Compiler das Zusammenpacken erzwingen

$$\frac{8+4+2+1}{1\ 1\ 1\ 1} = 15$$

C Run ▶

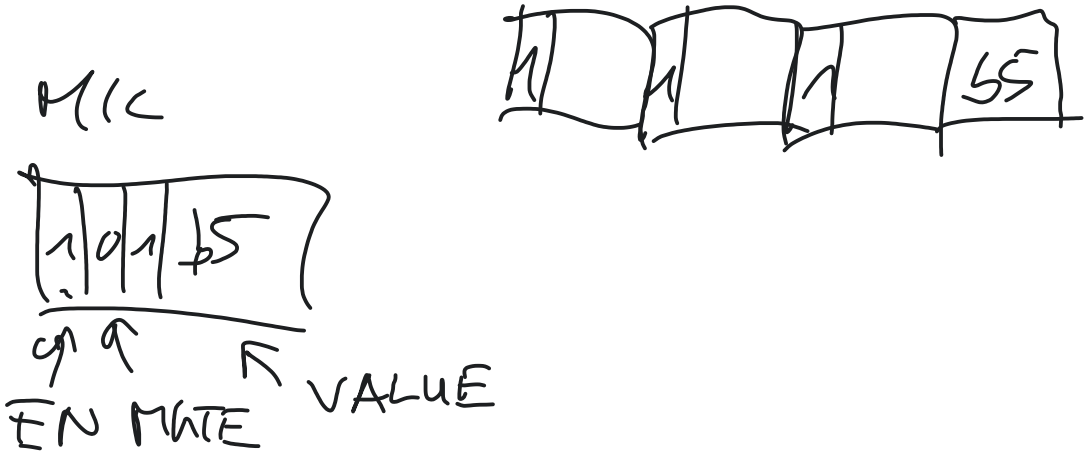
```c
#include <stdint.h>
#include <stdio.h>
struct Byte {
    uint8_t lower_half: 4;
    uint8_t upper_half: 4;
};

int main() {
    struct Byte b = { 0x47, 0x11};
    printf("Lower: %d, Upper: %d\n",
           b.lower_half, b.upper_half);
    return 0;
}
```



Wozu braucht man Bitfelder?

- Sind sehr nützlich in der hardwarenahen Programmierung
- Erlauben Abbildung von Konfigurationen auf Datenstrukturen
  - Konfigurationen sind bitcodiert, können direkt in Speicherregister geschrieben werden
  - Bessere Alternative zu mühseligen Bitoperationen
  - Compiler zur kompakten Speicherung zwingen `__attribute__((packed))`
    - Funktioniert für GCC- und Clang-Compiler
  - Für MSVC: `#pragma pack()`



Beispiel – Bitfelder für ein Mikrofon

C Run ▶

```c
#include <stdint.h>
#include <stdio.h>
#define MIC_CFG_REGISTER 0x4711

void write_register (size_t address, uint16_t value) { }

union {
    uint16_t value;
    struct {
        uint16_t enabled     :1;  // Bit 0
        uint16_t sensitivity :6;  // Bit 1-6
        uint16_t gain        :6;  // Bit 7-12
        uint16_t             :3;  // Leer, reserviert
    } __attribute__((packed));
} mic_cfg;

int main() {
    mic_cfg.value = 0; // Alles nullen (Init)
    mic_cfg.sensitivity = 0x20;
    mic_cfg.gain = 0x03;
```

Aufzählungen (*Enumerations*)

- Neben Strukturen und Unions weiteren selbstdefinierbaren Datentyp:

  Aufzählungen (*Enumerations*)
- Schlüsselwort `enum`
- Hauptverwendungszweck: Bündelung von Zuständen bzw. konstanten Werten
- Besser geeignet als `#define`

```
enum Animal { AFFE, KATZE, MAUS };

Animal a = KATZE;

switch (a) {
    case AFFE: break;
    case KATZE: break;
    case MAUS: break;
    default: break;
}
```

Repräsentation von Aufzählungen

- enum-Typen sind intern standardmäßig vom Typ `int`
- Nummerierung beginnt `0`:
- Folgendes ist daher identisch:

```
enum Animal { AFFE, KATZE, MAUS };
enum Animal { AFFE = 0, KATZE = 1, MAUS = 2 };
}
```

- Eigene Werte auch möglich, müssen aber explizit genannt werden:

```
enum Animal { AFFE = 0xAFFE, KATZE = 0xBAD, MAUS = 0xDEAD };
}
```

Repräsentation von Aufzählungen

- Seit C++11 bzw. C23: Statt `int` anderer zugrundeliegender Typ möglich

```
enum Animal: uint8_t { HUND, KATZE, MAUS };
```

- Alle Werte von `Animal` werden nun intern als `uint8_t` abgelegt
- `enum`-Variablen erlauben alle Operationen eines `int` 😕

→ Haben ebenfalls den gleichen Typ (genau wie bei `#define`)

Rechnen mit Aufzählungen

C Run ▶

```c
#include <stdio.h>

enum Animal {HUND, KATZE, MAUS};

int main() {
    enum Animal a = KATZE;
    a = a + 1;

    printf("Animal: %d\n", a);
    return 0;
}
```

Einschub: Enum-Klassen

- C++ versucht, diese Schwäche zu umgehen
- Einführung von neuem Typ `enum struct` bzw. `enum class`
- Intern immer noch `int`, aber keine implizite Umwandlung mehr möglich

  (Compiler verhindert dies)

```
enum struct Animal { AFFE, KATZE, MAUS };
Animal a = Animal::AFFE;
if (a == Animal::KATZE) { /* ... */ }
```

- Typauswahl nicht mehr wie in C, sondern mit Namespace-Präfix (→ später mehr)

Zwischenstand

- Bisher: selbstdefinierte Datentypen - `struct`, `union`, `enum`
- Was noch fehlt: Dynamisches Anlegen von Speicher
  - Bisher: Speicherbelegung bei Deklaration
  - Jetzt: Speicherbelegung zu beliebigen Zeitpunkten
- Anmerkung: Manuelle Verwaltung von Speicher → nächstes Kapitel

→ Hier nur die Grundlagen

Dynamische Speicherallokation

- Dynamisches Anlegen von Speicher (Allokation) ist wichtig

1. Strukturen können sehr groß sein
   - Problematisch als Funktionsparameter → muss immer kopiert werden
   - Größe des „Prograr
2. Daten sollen globa
   - Lokale Variablen hö
3. Speicher wird eventuell erst später benötigt
   - **Wir** bestimmen den Zeitpunkt
   - Wichtig bei limitierten Ressourcen

🚨 **Achtung: Speicherallokation eine der größten Stärken, gleichzeitig größte Schwäche**

Speicher anfordern

- Speicheranforderung in C mit `malloc()`
  - Speicher wird auf dem **Heap** angelegt (→ nächstes Kapitel)
  - Nach der Anlegung bleibt Speicher unbegrenzt belegt
  - **Explizite Freigabe** mit `free()` erforderlich
  - Sowohl `malloc()` als auch `free()` Teil von `stdlib.h`
  - Speicher bei Anlegung **nicht** initialisiert → Aufgabe des Programmierer:in

```c
#include <stdlib.h>
int main() {
void * memory = malloc(4);   // 4 Bytes angelegt
free(memory);                // Freigabe
    return 0;
}
```

Speicher anfordern

C Run ▶

```c
#include <stdlib.h>
int main() {
    void * memory = malloc(4);   // 4 Bytes angelegt
    free(memory);                // Freigabe
    return 0;
}
```

- Typ `void *` bezeichnet typischerweise nicht genauer definierten Speicher
  - Rückgabetyp von `malloc()`
  - Symbolisiert Speicher ohne Typ
- Mehr Details zu Speicher im nächsten Kapitel, nur das Wichtigste:
  - Stern-Operator * bedeutet, dass auf den Beginn eines Speicherblocks gezeigt wird (wie Index 0 eines Arrays)

Speicher anfordern

- Beispiel noch nicht besonders nützlich
  - Speicher wurde noch nicht befüllt
  - Hat noch unbekannten Typ `void *`
- Sinnvoller: Umwandlung des Rückgabewertes auf Zieltyp
- Beispiel:

```
char * addr = (char *) (malloc(sizeof(char) * 3));
```

- Operator `sizeof()` gibt Größe des angegebenen Objekts in Bytes zurück
  - Beispiel: `sizeof(char); // == 1`

Weitere Allokationsfunktionen

- Neben `malloc()` noch weitere Allokationsfunktionen verfügbar
- Rückgabewert ist Zeiger auf den Beginn eines Speicherbereichs
- `calloc()`:
  - Legt *Array* von Elementen an, kein manuelles Berechnen wie bei `malloc()` nötig
  - Alle Bytes des Speichers werden mit `0` initialisiert
  - Beispiel:

```c
int num = 5;
int * memory = (int *) calloc(num, sizeof(int)); // 5 * 4 Bytes, alle Wert 0
```

Weitere Allokationsfunktionen

- `realloc()`:
  - Verändert bereits angelegten Speicher
  - Startpunkt ist das erste Byte des vorhandenen Speichers
    - Vergrößerung → Hinzufügen an das Ende des vorhandenen Speichers (**ohne Initialisierung**)
    - Verkleinerung → „Abschneiden“ am Ende des Speichers
  - Beispiel:

```
void * memory = malloc(20);
memory = realloc(memory, 10);
```

Zwischenstand

- Bisher: Reihe von Funktionen zur Speicheranforderung
- Aber wie kann dieser Speicher befüllt werden?

→ Reihe von Speicheroperationen in `string.h`

`memcpy`, `memcmp`, `memmove`, `memset`

Operationen mit dynamisch angelegtem Speicher

- Für alle gilt: Bytes werden als `unsigned char` interpretiert
- `memcpy`: Kopiert `size` Bytes von Speicher `src` nach `dest`
  - `memcpy(dest, src, size);`
- `memmove`: Verschiebt `size` Bytes von Speicher `src` nach `dest`
  - `memmove(dest, src, size);`
- `memcmp`: Vergleicht die ersten `size` Bytes von Speicher `src` mit `dest`
  - `memcmp(dest, src, size);`
- `memset`: Kopiert das Zeichen `c` in die ersten `size` Bytes dest Speichers `dest`
  - `memset(dest, ch, size);`

Aufräumen ist wichtig!

- Wir können jetzt sowohl Speicher anlegen als auch beschreiben
- Eines fehlt noch: Freigabe von Speicher
- Manuelle Speicherverwaltung ist Fluch und Segen zugleich:
  - Der angeforderte Speicher **muss** wieder freigegeben werden
  - Freigabe liegt **vollständig in der Verantwortung der Programmierer:in** 🚨
  - **Sehr einfach**, dies versehentlich zu vergessen
- **Achtung**: Ist der Zeigerwert weg → Speicherort nicht mehr benutzbar
  - Geschieht automatisch bei Verlassen eines Code-Blocks
  - Konsequenz: Speicherleck (*Memory Leak*)

Speicherleck

- Einfaches Beispiel für ein Speicherleck:

C Run ▶

```c
#include <stdlib.h>

int main() {
    {
        void * memory =
        /* Viele Zeile
        // free(memory
    } // memory ab hie
    return 0;
}
```

Wenn möglich, für jedes `malloc()` direkt `free()`-Statement schreiben

- `free()` kann sehr leicht vergessen werden, besonders bei langen Funktionen

Fehlerbehandlung

- Abschließend: Was ist, wenn Speicher ausgeht?
- `malloc()` signalisiert Fehler über Rückgabewert
- Tritt ein Fehler auf, gibt `malloc()` `NULL` zurück

  → Nach dem Aufruf auf `NULL` prüfen und Fehlerbehandlung durchführen

```c
void *memory = malloc(42);
if (memory == NULL) {
    printf("Fehler!\n");
    exit(1);
}
```

Beispiel - Fehlerbehandlung von `malloc()`

C Run ▶

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

int main() {
    while (true) {
        void * memory = malloc(1024 * 1024); // 1 KiByte
        if (memory == nullptr) {
            break;
        }
    }
    printf("Speicher voll!\n");
    return 0;
}
```

Zusammenfassung

- Nicht alles mit elementaren Datentypen darstellbar
- *C* bietet drei Möglichkeiten, Datentypen selbst zu definieren
  - `struct` : Verbund von Datentypen
  - `union` : Vereinigung von Datentypen
  - `enum` : Aufzählung
- Dynamische Speicherverwaltung
  - erlaubt beliebigen Speicher zu beliebigen Zeitpunkten zu allozieren
  - Vollständig in der Hand der Programmierer:in
  - Freigabe ist zwingend erforderlich

Kapitel 08 - Eigene Datenstrukturen