Strukturen

>> Strukturen (**structs**) ermöglichen es, mehrere Variablen unterschiedlicher Datentypen in einem neuen Variablentyp zu vereinen, dem ein eigener Typname zugewiesen werden kann.

Wir nutzen Strukturen, um Elemente verschiedener Datentypen zu gruppieren, die logisch zusammengehören.

Eine Struktur ist wie eine „Super-Variable“.

Beispiel

```
struct car
{
    int year;
    char model[10];
    char plate[7];
    int odometer;
    double engine_size;
};
```

Wiederholung

Durch die Definition einer Struktur, die üblicherweise in separaten `.h`-Dateien oder am Anfang unserer Programme außerhalb von Funktionen erfolgt, haben wir einen neuen Datentyp erstellt.

Das bedeutet, wir können Variablen dieses Typs mit der gewohnten Syntax erstellen.

Der Zugriff auf die Felder (**Members**) der Struktur erfolgt mit dem Punkt-Operator (`.`).

Beispiel (Fortsetzung)

```
struct car
{
    int year;
    char model[10];
    char plate[7];
    int odometer;
    double engine_size;
};

struct car mycar;

mycar.year = 2011;
strcpy(mycar.plate, "CS50");
mycar.odometer = 50505;
```

Speicherung

Wie Variablen aller anderen Datentypen, müssen *structs* nicht zwangsläufig auf dem **Stack** angelegt werden.

Wir können sie zur Laufzeit auch dynamisch auf dem **Heap** allokieren, wenn unser Programm dies erfordert.

Um auf die Members einer *struct*, die auf dem Heap allokiert wurde, zuzugreifen, müssen wir zuerst den Pointer auf die Struktur **dereferenzieren.**

Neu: Speicherung auf dem Heap

```c
// auf dem Stack gespeichert
struct car mycar_stack;

mycar_stack.year = 2011;
strcpy(mycar_stack.plate, "CS50");
mycar_stack.odometer = 50505;


// auf dem Heap gespeichert
struct car *mycar = malloc(sizeof(struct car));

(*mycar).year = 2011;
strcpy((*mycar).plate, "CS50");
(*mycar).odometer = 50505;

// Klammern nötig, da . höhere Präzedenz hat als *
```

Verbesserung

Das ist etwas umständlich, aber es gibt einen kürzeren Weg.

Der Pfeil-Operator (−>) vereinfacht den Zugriff. Er führt zwei Operationen nacheinander aus:

Erst wird der Pointer auf der linken Seite des Operators dereferenziert.

Dann wird auf das Feld auf der rechten Seite des Operators zugegriffen

Neu: Speicherung auf dem Heap

```c
// auf dem Heap gespeichert
struct car *mycar = malloc(sizeof(struct car));

(*mycar).year = 2011;
strcpy((*mycar).plate, "CS50");
(*mycar).odometer = 50505;



// äquivalent, aber besser lesbar

mycar->year = 2011;
strcpy(mycar->plate, "CS50");
mycar->odometer = 50505;
```

EXTRAS IN 3 MINUTEN

FRAGEN – ANTWORTEN – RÄTSEL
UND KURZE ZUSAMMENFASSUNG

F1: Wenn man eine Struktur auf dem Stack anlegt, welchen Operator verwendet man für den Zugriff auf die Members?

F2: Wenn man eine Struktur auf dem Heap anlegt, welchen Operator verwendet man für den Zugriff auf die Members?

Alignment (Speicherausrichtung)

```c
// Array mit 10 int-Werten
int *werte = malloc(10 * sizeof(int));

// ... ist äquivalent zu:
int *werte  = malloc(40);

// Und wie viel Platz benötigt diese struct?
struct example {
    char flag;    // 1 Byte
    int value;    // 4 Byte
};

struct example *data = malloc(sizeof(struct example));

printf("%lu", sizeof(struct example)); // gibt 8 aus
```

Alignment (Speicherausrichtung)

```
// Daten werden im Speicher so ausgerichtet,
// dass sie an Adressen liegen, die ein Vielfaches
// der Wortbreite der CPU sind - also typischerweise
// 32 oder 64 Bit. Der Prozessor kann auf derart
// ausgerichtete Daten deutlich schneller zugreifen.


struct example {
    char flag;    // 1 Byte + 3 Byte für Padding!
    int value;    // 4 Byte
};

struct example *data = malloc(sizeof(struct example));

printf("%lu", sizeof(struct example)); // gibt 8 aus
```

Strukturen gruppieren verschiedene Datentypen in einem neuen Typ.

Feldzugriff mit Pfeil-Operator (->) bei Struktur-Pointern.

Definition global oder in Header-Dateien (.h).

Dynamische Allokation mit malloc() und sizeof().

Feldzugriff mit Punkt-Operator (.) bei Stack-Variablen.

Compiler fügt evtl. Padding für schnelleren Zugriff hinzu (Alignment) – dies erhöht den Platzbedarf.