

# Programmieren 1 – Informatik

## Strukturierte Datentypen

Prof. Dr.-Ing. Maike Stern | 22.11.2023

---

**SPENDEN-  
AKTION @ OTH**

**Soziales Projekt von Studierenden**  
am **06. und 07.12.** | 10 bis 14:30 Uhr  
im Studierendenhaus (Y006c)

**JETZT HELFEN!**

Illustration of three hands holding hearts: a green heart, a yellow heart, and a blue heart.

## CHECKLISTE

SPENDENAKTION @ OTH

### WAS WIRD GEBRAUCHT?

- **Rucksäcke**
- **Menstruationsartikel**
- Netto/Aldi etc. **Gutscheine**
- **Hoodies/Zipper**
- **Schuhe**
- Leggings, **Hosen**
- (lange) **Unterwäsche, Socken**
- **SIM-Karten**
- Zelte, **Schlafsäcke**, Isomatten



Es werden vor allem  
**Männerklamotten** in  
Größe M-XL benötigt

- Pointer
- Pointerarithmetik & Arrayindexierung
- Strings
- Call-by-Value & Call-by-Reference

Pointer		
int *p = NULL;	Deklaration mit Initialisierung auf NULL	
p = &i;	Initialisierung	
*p;	Dereferenzierung (Wert von i)	
p;	Adresse, auf die p zeigt	
&p;	Adresse von p	
Auf Arrayelemente zugreifen		
*pointer	*array	erstes Element des Arrays
pointer[0]	array[0]	
*(pointer+n)	*(array+n)	Zugriff auf <i>n</i> tes Element
pointer[n]	array[n]	
pointer	array	Zugriff auf die Adresse des ersten Elements
&pointer[0]	&array[0]	
pointer+n	array+n	Zugriff auf die Adresse des <i>n</i> ten Elements
&pointer[n]	&array[n]	

- Call-by-Reference bezeichnet die Übergabe von Funktionsparameter per Referenz
- Das heißt, im Gegensatz zu Call-by-Value werden die Parameterwerte nicht in die Funktion hineinkopiert, sondern es wird die Adresse zu den Werten übergeben (wobei die Adresse in die Funktion kopiert wird)
- Dadurch belegen Werte nicht doppelt Speicherplatz → insbesondere bei großen Daten sinnvoll

## 1. Variante

- Die Variablen a, b und c werden in der main-Funktion deklariert und initialisiert
- Funktionsparameter: Variablen bzw. Pointer  
Rückgabewert: void
- Funktionsargumente: Die Werte bzw. Adressen der Variablen

```
void addNumbers(int a, int b, int *result){  
    *result = a + b;  
}  
  
int main(){  
    int a = 2, b = 3;  
    int c = 0;  
  
    addNumbers(a, b, &c);  
}
```

↑  
Variablenwerte, die nicht verändert  
werden, immer als Werte

## 2. Variante

- Die Variablen a und b werden in der main-Funktion deklariert und initialisiert
- Es wird Heapspeicher für eine Integer-Variablen reserviert
- Funktionsparameter: Variablen bzw. Pointer  
Rückgabewert: void
- Funktionsargumente: Die Variablenwerte sowie der Pointer, der auf den Heapspeicher zeigt
- Das Ergebnis wird dann auf dem Heap gespeichert

```
void addNumbers(int a, int b, int *result){  
    *result = a + b;  
}  
  
int main(){  
    int a = 2, b = 3;  
    int *p = (int*) malloc(sizeof(int));  
  
    addNumbers(a, b, p);  
}
```

## 3. Variante

- Die Variablen a und b werden in der main-Funktion deklariert und initialisiert
- Funktionsparameter: Variablenwerte  
Rückgabewert: Pointer
- Funktionsargumente: Variablenwerte
- In der Funktion wird Speicher auf dem Heap reserviert für eine Integer
- Das Ergebnis wird in den Heapspeicher geschrieben
- Die Funktion gibt den Pointer auf den Heapspeicher zurück

```
int *addNumbers2(int a, int b) {  
    int *result = (int*) malloc(sizeof(int));  
    *result = a + b;  
    return result;  
}  
  
int main(){  
    int a = 2, b = 3;  
    int *ip;  
  
    ip = addNumbers2(a, b);  
}
```

```
#include <stdio.h>

int main(){
    int i = 0;
    int p = NULL;

    p = i;
    *p = 23;

    printf("Der neue Wert von i ist: %d", i);
    return 0;
}
```



- Pointer
- Pointerarithmetik & Arrayindexierung
- Call-by-Value & Call-by-Reference
- Null- & Void-Pointer

void-Pointer Typumwandlung		Beispiel
<code>void *p = NULL;</code>	Deklaration	<code>void *p = NULL; int i = 10;</code>
<code>p = (typ *) &amp;i;</code>	Typumwandlung & Initialisierung	<code>p = (int *) &amp;i;</code>
<code>*(typ *)p = wert;</code>	Dereferenzierung	<code>*(int *)p = 22;</code>
<code>var = *(typ *)p;</code>	Dereferenzierung	<code>int x = *(int *)p;</code>

- Pointer
- Pointerarithmetik & Arrayindexierung
- Call-by-Value & Call-by-Reference
- Null- & Void-Pointer
- Stack- und Heapspeicher
- malloc, calloc, realloc
- Pointer auf Pointer
- Funktionspointer

Heapspeicher allokieren und freigeben	
<code>int *p = NULL</code>	Pointerdeklaration
<code>p = (int *) malloc(sizeof(int));</code>	Reserviert Heapspeicher für eine Integer
<code>p = (int *) calloc(10, sizeof(int));</code>	Reserviert Heapspeicher für einen Block von zehn Integer & initialisiert mit 0
<code>p = (int *) realloc(p, newSize);</code>	Vergrößert oder verkleinert den reservierten Heapspeicher
<code>free(p)</code>	Gibt den reservierten Heapspeicher frei

- Typedef
- Strukturierte Datentypen
- Enumerationen
- Unions

- Im Verlauf der Vorlesung haben wir verschiedene Datentypen kennengelernt, z.B. integer, float, double, unsigned long integer, pointer, arrays

Adresse	Typ	Name	Wert
0x16ee1ae80	int	i	10
0x16ee1ae7c	float	f	2.178
0x16ee1ae78	int	a[]	{3, 54, 862, 1}
0x16ee1ae52	int *	p	0x16ee1ae80

- Im Verlauf der Vorlesung haben wir verschiedene Datentypen kennengelernt, z.B. integer, float, double, unsigned long integer, pointer, arrays
- Außerdem haben wir uns Typumwandlungen (typecasting) angesehen, also das Wandeln einer Variablen mit einem bestimmten Variablentyp in einen anderen
  - implizite Typumwandlung  
Der Compiler wandelt den Datentyp einer Variablen selbstständig um
  - explizite Typumwandlung  
Explizite Typumwandlung durch den/die Programmierer:in

Adresse	Typ	Name	Wert
0x16ee1ae80	int	i	10
0x16ee1ae7c	float	f	2.178
0x16ee1ae78	int	a[]	{3, 54, 862, 1}
0x16ee1ae52	int *	p	0x16ee1ae80

result2: 2.5

Explizite Typumwandlung der Integer-variablen x in eine float, wodurch die Division ohne Informationsverlust durchgeführt wird

result1: 2.0

Der Compiler wandelt das Divisionsergebnis (2) in eine float um

```
#include <stdio.h>

int main(){
    int x = 5, y = 2;
    float result1 = x / y;
    float result2 = (float) x / y;
}
```

- Im Verlauf der Vorlesung haben wir verschiedene Datentypen kennengelernt, z.B. integer, float, double, unsigned long integer, pointer, arrays
- Außerdem haben wir uns Typumwandlungen (typecasting) angesehen, also das Wandeln einer Variablen mit einem bestimmten Variablentyp in einen anderen
  - implizite Typumwandlung  
Der Compiler wandelt den Datentyp einer Variablen selbstständig um
  - explizite Typumwandlung  
Explizite Typumwandlung durch den/die Programmierer:in
- Und wir haben dynamische Arrays (also Pointer auf einen manuell verwalteten Speicherbereich) kennengelernt, als Möglichkeit die Arraygröße zur Laufzeit zu ändern

Adresse	Typ	Name	Wert
0x16ee1ae80	int	i	10
0x16ee1ae7c	float	f	2.178
0x16ee1ae78	int	a[]	{3, 54, 862, 1}
0x16ee1ae52	int *	p	0x16ee1ae80

```
int main(){
    int *values = NULL, numElements = 0, i = 0;

    printf("\nAnzahl der Werte, die eingegeben werden sollen: \n");
    scanf("%d", &numElements);

    values = (int *) malloc(numElements * sizeof(int));
}
```

# Typedef

Typedef ist ein Schlüsselwort, mit dem Schlüsselbegriffen ein Synonym zugewiesen werden kann

- Syntax

`typedef Datentyp neueBezeichnung;`



Der Name des neuen Bezeichners kann frei gewählt werden, unter Berücksichtigung der Variablennamenskonventionen (keine Zahlen am Anfang, keine Leerzeichen oder Sonderzeichen, ...)

- Im Codebeispiel wird dem Datentyp Integer ein neuer Begriff zugeordnet

Integer können jetzt entweder klassisch mit *int* deklariert werden, oder mit dem neuen Bezeichnet *myInt*

Datentyp, der dann auch mit der neuen Bezeichnung verwendet werden kann

```
int main(){  
    typedef int myInt;  
  
    myInt i = 7;  
    int j = 8;  
  
    i = 5;  
  
    return 0;  
}
```



bei Arrays wird die eckige  
Klammer an den neuen  
Bezeichner gehängt

Typedef  
Arrays

Typedef ist ein Schlüsselwort, mit dem C-Schlüsselbegriffen ein Synonym zugewiesen werden kann

- Syntax

`typedef Datentyp neueArrayBezeichnung[];`

Datentyp, der eine zusätzliche  
Bezeichnung erhalten soll

eckige Arrayklammern, mit  
oder ohne Dimensionsangabe

- Im Codebeispiel wird ein neuer Bezeichner für Arrays mit drei Elementen deklariert
- Natürlich ist es auch möglich, Arrays beliebiger Größe einen zusätzlichen Namen zu geben
- Typedefs sollten mit Bedacht eingesetzt werden, da sie den eigentlichen Datentyp bzw. die Größe des Arrays verschleiern. Sinnvolle Einsatzmöglichkeit: **Structs**

```
int main(){  
  
    typedef int threeElementsArray[3];  
    typedef char string[100];  
  
    threeElementsArray a = {1,2,3};  
    string s1 = "Hello, World!",  
    string s2 = "I love computer \\  
                science!";  
  
    a[0] = 8;  
    strcat(s1, " ");  
    strcat(s1, s2);  
  
    return 0;  
}
```

Mit typedef kann auch die Notation von Funktionspointern vereinfacht werden

Die neue Bezeichnung, die durch typedef festgelegt wird, ist der Name des Funktionspointers

Deklaration eines Funktionspointers, der auf eine Funktion wie printf / scanf zeigen kann

ohne typedef

```
int main(){  
    int (* ptr) (const char *, ...);  
    ptr = printf;  
    (* ptr) ("Bitte Wert: \n");  
    return 0;  
}
```

Um den typedef-Funktionspointer zu verwenden muss ein Funktionspointer vom erstellten Typ deklariert werden und kann dann ohne zusätzliche Notation verwendet werden

```
typedef int (* FunctionPointer) (const char *, ...);  
  
int main(){  
    int value = 0;  
    FunctionPointer function_pointer;  
  
    function_pointer = printf;  
    function_pointer ("Bitte Wert eingeben: \n");  
  
    function_pointer = scanf;  
    function_pointer ("%d", &value);  
  
    return 0;  
}
```

# Strukturen (Structs)

Structs sind eine Möglichkeit mehrere Datentypen strukturiert zu einem neuen Datentyp zu kombinieren

```
struct newDataType {  
    int i;  
    int j;  
    float f;  
    array a[];  
};
```

- In den vorherigen Vorlesungen haben wir uns Arrays angeschaut, als Möglichkeit, mehrere Werte in einer Variablen zu speichern
- Arrays umfassen jedoch immer nur einen Datentypen, Integer-Arrays oder Strings (Character-Arrays) zum Beispiel
- Strukturen (structs) ermöglichen es, verschiedene Datentypen in einer Datenstruktur zu vereinen und somit einen neuen Datentyp zu kreieren
- Structs sind also Datenblöcke, in denen man Variablen mit verschiedenen Datentypen kapseln kann → Strukturiert das Programm

struct s

char a[5]										float b								int c			

Strukturen werden mittels *struct* erstellt:

```
struct typeName {  
    Datentyp1 varName1;  
    Datentyp2 varName2;  
    ...  
    DatentypN varNameN;  
};
```

typeName ist der Name der neuen Strukturvariablen. Die typeName-Struktur kann wie ein Datentyp für verschiedene Variablen verwendet werden

Semikolon nicht vergessen!

Deklaration :

```
struct typeName structName;
```

Hier wird eine neue Strukturvariable vom Typ typeName mit dem Namen structName erstellt

Die Wertzuweisung erfolgt mittels Punktoperator

```
structName.varName1 = 1337;  
...
```

Typ (keyword)	Name
int	x
char	c
struct typeName	structName

Strukturen werden mittels *struct* erstellt:

```
struct typeName {  
    Datentyp1 varName1;  
    Datentyp2 varName2;  
    ...  
    DatentypN varNameN;  
};
```

Deklaration :

```
struct typeName structName;  
  
structName.varName1 = 1337;  
...
```

Übung:

Erstellen Sie eine struct-Variable names StudentData, die als Strukturvariablen den Vornamen, Nachnamen und die StudentID (also Matrikelnummer) enthält.

Deklarieren Sie die struct und weisen Sie ihr Werte für eine/n Student:in zu

In der Regel werden Structs wie Funktionen vor der main-Funktion erstellt. Es gibt aber auch Anwendungsfälle, in denen eine Deklaration innerhalb der main-Funktion sinnvoll ist

Die Deklaration der einzelnen structs erfolgt dann innerhalb der main-Funktion

```
#include <stdio.h>

struct studentData {
    char firstName[20];
    char lastName[20];
    unsigned int studentID;
};

int main(){
    // Struktur vom Typ studentData mit
    // dem Strukturnamen student1
    struct studentData student1;

    strcpy(student1.firstName, "Maike");
    strcpy(student1.lastName, "Stern");
    student1.studentID = 1234;

    return 0;
}
```

Frage:  
Warum wird der String mittels der strcpy-Funktion initialisiert?  
Was wäre eine alternative Implementierung?



Es gibt verschiedene Möglichkeiten, wie Structs deklariert und initialisiert werden können:

```
struct typeName {
    datentyp var1;
    datentyp var2;
    ...
}

int main(){
    struct typeName structName1 = {value, "string", ...};
    struct typeName structName2 = {.var2 = "string", ...};
    ...}
```

Initialisierung bei der Deklaration

Die Werte werden in geschweiften Klammern in der entsprechenden Reihenfolge angegeben. Sollen nur einzelne Variablen initialisiert werden, geht dies mittels Punktoperator.

```
struct typeName {
    datentyp var1;
    datentyp var2;
    ...
}

int main(){
    struct typeName structName;
    structName.varName = value;
    strcpy(structName.varName, "string");
    ...}
```

Deklaration und Initialisierung getrennt.

Die Initialisierung erfolgt über den Punktoperator. Ähnlich zu: `x = 10;` nur das hier der Name der Struktur vorne angestellt wird: `studentData.studentID = 123;`

<pre>struct typeName {     datentyp var1;     datentyp var2;     ... };  int main(){     struct typeName structName;     ...}</pre>	<pre>struct typeName {     datentyp var1;     datentyp var2;     ... };  typedef struct alias structName;  int main(){     alias structName;     ...}</pre>	<pre>typedef struct typeName {     datentyp var1;     datentyp var2;     ... } alias;  int main(){     alias structName;     ...}</pre>
Deklaration ohne typedef	Deklaration mit typedef I: <ul style="list-style-type: none"> <li>• Struct erstellen</li> <li>• Mittels typedef einen neuen Alias für die Struktur zuweisen</li> </ul>	Deklaration mit typedef II: <p>Typedef kann auch direkt bei der Deklaration der Struktur verwendet werden</p>

## Übung:

1. Erstellen Sie einen neuen Bezeichner für float-Variablen mittels typedef, deklarieren und initialisieren Sie eine entsprechende Variable.
2. Erstellen Sie eine struct-Variable names StudentData mit dem Alias (ebenfalls) studentData, die als Strukturvariablen den Vornamen, Nachnamen und die StudentID enthält. Deklarieren Sie die struct und weisen Sie ihr Werte für eine/n Student:in zu.

```
struct typeName {
    datentyp var1;
    datentyp var2;
    ...
};
```

```
typedef struct alias structName;
```

```
int main(){
    alias structName;
    ...}
```

Deklaration mit typedef I:

- Struct erstellen
- Mittels typedef einen neuen Alias für die Struktur zuweisen

```
typedef struct typeName {
    datentyp var1;
    datentyp var2;
    ...
} alias;
```

```
int main(){
    alias structName;
    ...}
```

Deklaration mit typedef II:

Typedef kann auch direkt bei der Deklaration der Struktur verwendet werden

# Structs in Funktionen

Structs können sowohl Call-by-Value als auch Call-by-Reference an Funktionen übergeben werden

- Call-by-Value

Frage:

Warum ist es oft eine schlechte Idee, structs per Call-by-Value zu übergeben?

```
typedef struct studentData{
    char firstName[20];
    char lastName[20];
    unsigned int studentID;
} studentData;

void printStruct(studentData student){
    printf("\nThe student's name is %s %s\n", \
student.firstName, student.lastName);
    printf("%s's ID is %d\n\n", \
student.firstName, student.studentID);
}

int main(){
    studentData student1 = {"Maike", "Stern", \
1234};

    printStruct(student1);
    return 0;
}
```

Structs können sowohl Call-by-Value als auch Call-by-Reference an Funktionen übergeben werden

- Call-by-Value

Werden structs Call-by-Value an eine Funktion übergeben, so wird im Stackframe der Funktion eine Kopie der vollständigen struct angelegt

→ kann bei häufigen Funktionsaufrufen und / oder umfangreichen Strukturen die Laufzeit des Programms erhöhen

```
typedef struct studentData{
    char firstName[20];
    char lastName[20];
    unsigned int studentID;
} studentData;

void printStruct(studentData student){
    printf("\nThe student's name is %s %s\n", \
        student.firstName, student.lastName);
    printf("%s's ID is %d\n\n", \
        student.firstName, student.studentID);
}

int main(){
    studentData student1 = {"Maike", "Stern", \
        1234};

    printStruct(student1);
    return 0;
}
```

Structs können sowohl Call-by-Value als auch Call-by-Reference an Funktionen übergeben werden

- Call-by-Value

- Call-by-Reference

Wie bei anderen Variablen auch, können Structs Call-by-Reference übergeben werden. Das heißt, der Funktion wird ein Pointer übergeben, der auf die Anfangsadresse der Struct zeigt

```
typedef struct studentData{
    char firstName[20];
    char lastName[20];
    unsigned int studentID;
} studentData;

void printStruct(studentData *studentPointer){
    ...
}

int main(){
    studentData student1 = {"Maike", "Stern", \
1234};

    printStruct(&student1);
    return 0;
}
```

Anders als bei primitiven Datentypen wird bei der Dereferenzierung nicht ein Sternchen (\*) sondern ein Pfeiloperator verwendet (->)

Alternative:  
(studentData\*)(studentPointer.firstName)

```
typedef struct studentData{
    char firstName[20];
    char lastName[20];
    unsigned int studentID;
} studentData;

void printStruct(studentData *studentPointer){

    printf("\nThe student's name is %s %s\n", \
        studentPointer -> firstName, \
        studentPointer -> lastName);
    printf("ID: %d\n\n", \
        studentPointer -> studentID);
}

int main(){
    studentData student1 = {"Maike", "Stern", \
        1234};

    printStruct(&student1);
    return 0;
}
```

Deklaration eines  
Struct-Pointers:  
Datentyp \*Name

Übergabe der  
Anfangsadresse der  
struct an die Funktion



Sollen Funktionen Structs zurückgeben, so kann

- entweder die Struct selbst als Rückgabewert zurückgegeben werden (nicht empfehlenswert)
- oder ein Pointer auf die Struktur zurückgegeben werden

Sollen Funktionen Structs zurückgeben, so kann

- entweder die Struct selbst als Rückgabewert zurückgegeben werden (nicht empfehlenswert)
- oder ein Pointer auf die Struct zurückgegeben werden

in main():

- Pointer vom Typ der entsprechenden Struct in main() anlegen
- Rückgabewert der Funktion in den Pointer schreiben

in der Funktion:

- Pointer vom Typ der entsprechenden Struct anlegen
- Heap-Speicherplatz für die Struct reservieren mittels malloc()
- Struct bearbeiten
- Struct-Pointer zurückgeben

```
typedef struct studentData{  
    char firstName[20];  
    char lastName[20];  
    unsigned int studentID;  
} studentData;  
  
studentData *inputStructData(){  
    studentData *student;  
    student = (studentData*)malloc(sizeof(studentData));  
  
    printf("Bitte Vornamen eingeben: \n");  
    fgets(student->firstName, SIZE, stdin);  
    return student;  
}  
  
int main(){  
    studentData *student;  
  
    student = inputStructData();  
    return 0;  
}
```

Deklaration eines  
Structpointers

Auf dem Heap wird  
Speicherplatz für die Struct  
angelegt

Die Funktion gibt den  
Structpointer zurück, der  
auf die Anfangsadresse der  
struct im Heapspeicher  
zeigt

Rückgabebetyp der Funktion:  
Structpointer

Wie zuvor wird auf die  
einzelnen  
Strukturvariablen mittels  
Pfeiloperator zugegriffen

Zugriff auf Structvariablen:

- `student.studentID = 123;`
- `strcpy(student.firstName, "name");`

Zugriff auf Structvariable per Pointer:

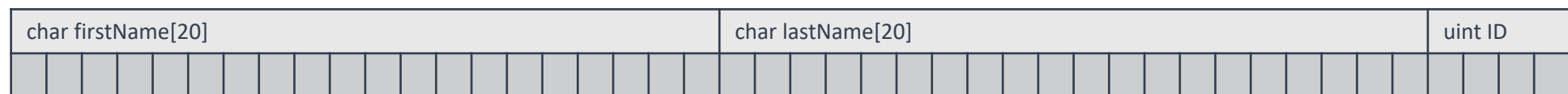
- `student -> studentID = 123;`
- `strcpy(student -> firstName, "name");`

```
typedef struct studentData{  
    char firstName[20];  
    char lastName[20];  
    unsigned int studentID;  
} studentData;  
  
int main(){  
  
    studentData studentVar;  
    studentData *studentPointer;  
  
    return 0;  
}
```

`structName.var`



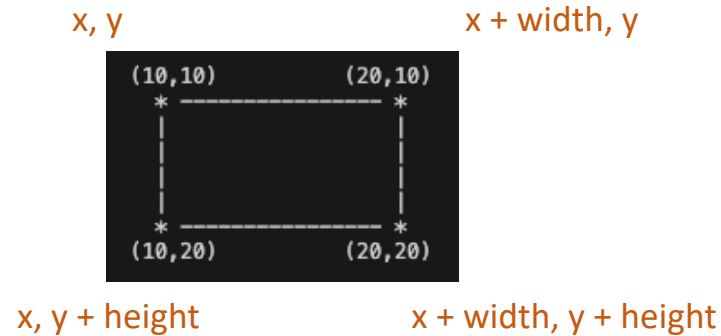
`structName -> var`



## Programm zum Ausdrucken eines Rechtecks mit Koordinatenangaben

Structs:

- coordinates
- dimension
- rectangle



Structs können auch  
verschachtelt werden  
(nested structures)

```
typedef struct coordinates{
    int x;
    int y;
} coordinates;

typedef struct dimension {
    int width;
    int height;
} dimension;

typedef struct rectangle {
    coordinates position;
    dimension dimension;
} rectangle;

int main(){
    ...
    return 0;
}
```

## Nested structures / verschachtelte Strukturen

Struct für die Uhrzeit, mit Integervariablen für Stunden und Minuten

Struct, die ein Character-Array sowie eine struct vom Typ time als Variablen hat

Deklaration einer Struct vom Typ meeting namens dataScienceMeetup. Die Struct wird direkt initialisiert mit den Variablenwerten. Die Variablenwerte der Struct time werden in geschweiften Klammern übergeben

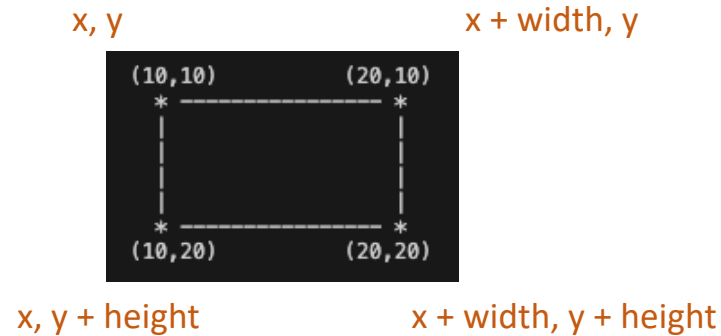
Zugriff auf die Werte erfolgt über den Punktoperator

```
typedef struct Time {  
    unsigned int hour;  
    unsigned int minutes;  
} Time;  
  
typedef struct Meeting {  
    char name[20];  
    Time time;  
} Meeting;  
  
int main() {  
    Meeting dataScienceMeetup = {"Data Science MeetUp", {18, 30}};  
  
    printf("The %s is at %d:%d", dataScienceMeetup.name,  
        dataScienceMeetup.time.hour,  
        dataScienceMeetup.time.minutes);  
  
    return 0;  
}
```

## Programm zum Ausdrucken eines Rechtecks mit Koordinatenangaben

Structs:

- coordinates
- dimension
- rectangle



Die verschachtelte Struktur könnte also folgendermaßen initialisiert werden:  
rectangle rect = {{10, 10}, {10, 10}};

Zugriff:  
printf("X is %d", rect.position.x);

```
typedef struct coordinates{
    int x;
    int y;
} coordinates;

typedef struct dimension {
    int width;
    int height;
} dimension;

typedef struct rectangle {
    coordinates position;
    dimension dimension;
} rectangle;

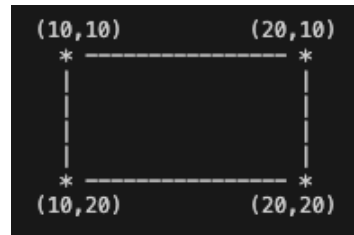
int main(){
    ...
    return 0;
}
```

Programm zum Ausdrucken eines Rechtecks mit Koordinatenangaben

Structs:

- coordinates
- dimension
- rectangle

$x, y$   $x + \text{width}, y$



Funktionen:

$x, y + \text{height}$   $x + \text{width}, y + \text{height}$

- `rectangle* create_rectangle (coordinates upperLeft, coordinates lowerRight)`
- `void print_rectangle (rectangle *rect)`
- `void move_rectangle (rectangle *rect, coordinates newPosition)`
- `void resize_rectangle (rectangle *rect, dimension newSize)`



`rectangle* create_rectangle (coordinates upperLeft,  
coordinates lowerRight)`

- Der Funktion werden zwei Structvariablen übergeben, einmal die Koordinaten der oberen linken Ecke und einmal die Koordinaten der unteren rechten Ecke

```
typedef struct coordinates{  
    int x;  
    int y;  
} coordinates;
```

- Innerhalb der Funktion werden die Rechtecksdimensionen bestimmt...
- ... und dann in eine Struct vom Typ rectangle geschrieben, für die zuvor Heapspeicher reserviert wurde
- Die Funktion gibt einen Structpointer vom Typ rectangle aus, der auf die Struct im Heapspeicher zeigt

```
rectangle* create_rectangle(coordinates upperLeft,  
coordinates lowerRight) {  
  
    // input validation  
    if (upperLeft.x > lowerRight.x) {  
        return NULL;  
    }  
  
    if (upperLeft.y > lowerRight.y) {  
        return NULL;  
    }  
  
    const dimension dim = {  
        .width = lowerRight.x - upperLeft.x,  
        .height = lowerRight.y - upperLeft.y  
    };  
  
    rectangle* result =  
        (rectangle*)malloc(sizeof(rectangle));  
    result->position = upperLeft;  
    result->dimension = dim;  
  
    return result;  
}
```

Programm zum Ausdrucken eines Rechtecks mit Koordinatenangaben

Structs:

- coordinates, dimension, rectangle

Funktionen:

- `rectangle* create_rectangle (coordinates upperLeft, coordinates lowerRight)`
- `void print_rectangle (rectangle *rect)`
- `void move_rectangle (rectangle *rect, coordinates newPosition)`
- `void resize_rectangle (rectangle *rect, dimension newSize)`

*move\_rectangle* und *resize\_rectangle* übernehmen einen Zeiger auf die Struct im Heapspeicher und ändern im Heap die Koordinaten- bzw. die Größenangaben

```
typedef struct dimension {  
    int width;  
    int height;  
} dimension;
```

```
void resize_rectangle(rectangle* rect,  
    dimension newSize) {  
  
    rect->dimension = newSize;  
}
```

## Übung:

Erstellen Sie ein Programm mit folgenden Structs:

- *Datum* mit den Variablen Stunde & Minute & Wochentag
- *Vorlesung* mit den Variablen Vorlesungsname & *Datum* (also die erste Struct)

Initialisieren Sie die Struct *Vorlesung* in *main()* und lassen Sie die Werte ausgeben.

```
struct typeName {  
    datentyp var1;  
    datentyp var2;  
    ...  
};  
  
typedef struct alias structName;  
  
int main(){  
    alias structName;  
    ...}
```

Deklaration mit typedef I:

- Struct erstellen
- Mittels typedef einen neuen Alias für die Struktur zuweisen

```
typedef struct typeName {  
    datentyp var1;  
    datentyp var2;  
    ...  
} alias;  
  
int main(){  
    alias structName;  
    ...}
```

Deklaration mit typedef II:

Typedef kann auch direkt bei der Deklaration der Struktur verwendet werden

Genau wie primitive Variablen können mehrere Structs in einem Array gesammelt werden

```
typedef struct Student{
    char firstName[20];
    char lastName[20];
    unsigned int studentID;
} Student;
```

```
Student student[100];
```

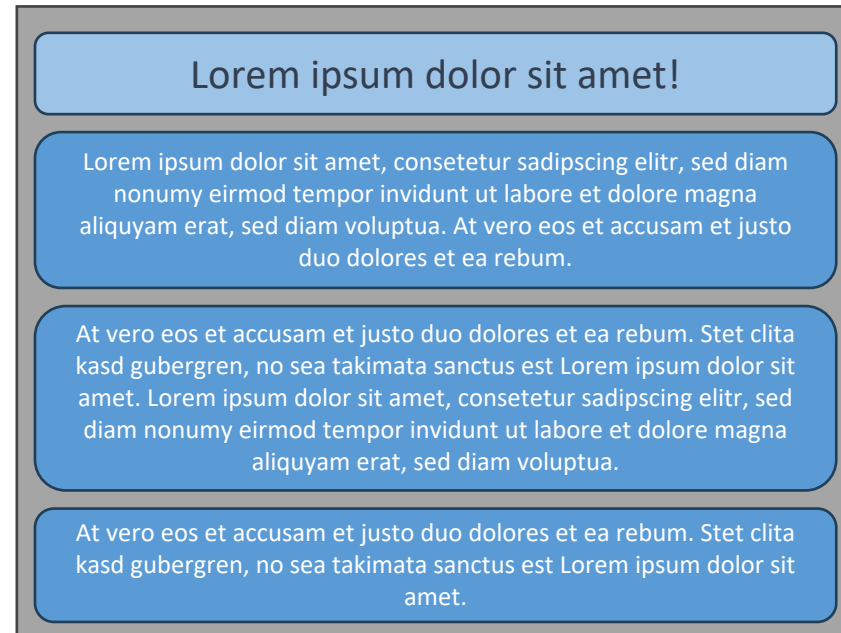
Im Speicher wird Platz für 100 Strukturen  
vom Typ students reserviert

student[0]	char firstName[20]	char lastName[20]	uint ID
	<div></div>	<div></div>	<div></div>
student[...]	char firstName[20]	char lastName[20]	uint ID
	<div></div>	<div></div>	<div></div>
student[n]	char firstName[20]	char lastName[20]	uint ID
	<div></div>	<div></div>	<div></div>

## Nested Structs, Pointer, Struct Arrays: Codebeispiel

### Programm zum Befüllen einer Webseite

- Header  
Überschrift der Webseite
- Content  
Variable für den Fließtext
- Body  
Inhalt der Webseite
- Payload  
Fasst alles zusammen



```
#define MAX 10

typedef struct Header {
    char name[20];
} Header;

typedef struct Content {
    char text[255];
} Content;

typedef struct Body {
    int num_element;
    Content contents[MAX];
} Body;

typedef struct Payload {
    Header header;
    Body body;
} Payload;
```

## Nested Structs, Pointer, Struct Arrays: Codebeispiel

## Programm zum Befüllen einer Webseite

- Header  
Überschrift der Webseite
- Content  
Variable für den Fließtext
- Body  
Inhalt der Webseite
- Payload  
Fasst alles zusammen

Die Struct-Variable  
Content wird als Array  
angelegt

content[0]	char name[20]
content[...]	char name[20]
content[9]	char name[20]

```
#define MAX 10

typedef struct Header {
    char name[20];
} Header;

typedef struct Content {
    char text[255];
} Content;

typedef struct Body {
    int num_element;
    Content contents[MAX];
} Body;

typedef struct Payload {
    Header header;
    Body body;
} Payload;
```

## Nested Structs, Pointer, Struct Arrays: Codebeispiel

```
int main() {  
    Body body = {.num_element = 0};  
  
    add_content_to_body("This is the first  
                        string...", &body);  
    add_content_to_body("... and this is the  
                        second one", &body);  
  
    Header header = {  
        .name = "Some serious stuff"  
    };  
  
    Payload payload = {  
        .body = body,  
        .header = header  
    };  
  
    print_payload(&payload);  
    return 0;  
}
```

Deklariert und initialisiert die Body-Struct mit Werten

Deklariert und initialisiert die Header-Struct mit Werten

Deklariert und initialisiert die Payload-Struct mit den eben erstellten Structs

Übergibt die Payload-Struct an die Funktion print\_payload um die Werte auszugeben

## Nested Structs, Pointer, Struct Arrays: Codebeispiel

```
int main() {
    Body body = {.num_element = 0};

    add_content_to_body("This is the first
                        string...", &body);
    add_content_to_body("... and this is the
                        second one", &body);

    Header header = {
        .name = "Some serious stuff"
    };

    Payload payload = {
        .body = body,
        .header = header
    };

    print_payload(&payload);
    return 0;
}
```

Body ist die Struct-Variable, die den Inhalt (content) der Webseite aufnimmt, also die Fließtexte.

Es können MAX Textflächen mit Fließtext gefüllt werden, über die Content-Struct. Num\_element zählt, wieviele Fließtexte (Element) das Array contents tatsächlich enthält.

Um Text zur Body-Struct hinzuzufügen gibt es die Funktion add\_content\_to\_body.

Zunächst muss aber num\_element mit 0 initialisiert werden, da sonst undefiniertes Verhalten auftreten kann.

```
#define MAX 10

typedef struct Content {
    char text[255];
} Content;

typedef struct Body {
    int num_element;
    Content contents[MAX];
} Body;
```



## Nested Structs, Pointer, Struct Arrays: Codebeispiel

```
int main() {  
    Body body = {.num_element = 0};  
  
    add_content_to_body("This is the first  
                        string...", &body);  
    add_content_to_body("... and this is the  
                        second one", &body);
```

```
    Header header = {  
        .name = "Some serious stuff"  
    };  
  
    Payload payload = {  
        body = body
```

```
#define MAX 10
```

```
typedef struct Content {  
    char text[255];  
} Content;
```

```
typedef struct Body {  
    int num_element;  
    Content contents[MAX];  
} Body;
```

```
void add_content_to_body(char* text, Body* body) {  
    strcpy(body->contents[body->num_elements].text, text);  
    body->num_elements++;  
}
```

Die Funktion `add_content_to_body` nimmt den Fließtext auf sowie einen Pointer auf die `body`-Struct, die zuvor in `main()` deklariert wurde.

Innerhalb der Funktion wird dem `contents`-Array in der `body`-Struct der Text zugewiesen.

Außerdem wird die Anzahl der verwendeten Element im Array mittels `num_element` hochgezählt

## Nested Structs, Pointer, Struct Arrays: Codebeispiel

```
int main() {
    Body body = {.num_element = 0};

    add_content_to_body("This is the first
                        string...", &body);
    add_content_to_body("... and this is the
                        second one", &body);

    Header header = {
        .name = "Some serious stuff"
    };

    Payload payload = {
        .body = body,
        .header = header
    };

    print_payload(&payload);
    return 0;
}
```

```
void add_content_to_body(char* text, Body* body) {
    strcpy(body->contents[body->num_elements].text, text);
    body->num_elements++;
}
```

### Achtung!

Um einem Struct-Array Werte zuzuweisen, muss der Arrayindex richtig gesetzt werden:

`body -> contents[body -> num_elements].text`

### Generell:

```
structName[element].varName = 123;
strcpy(structName[element].varName, "hallo");
```

```
#define MAX 10

typedef struct Content {
    char text[255];
} Content;

typedef struct Body {
    int num_element;
    Content contents[MAX];
} Body;
```

## Nested Structs, Pointer, Struct Arrays: Codebeispiel

```
int main() {
    Body body = {.num_element = 0};

    add_content_to_body("This is the first
                        string...", &body);
    add_content_to_body("... and this is the
                        second one", &body);

    Header header = {
        .name = "Some serious stuff"
    };

    Payload payload = {
        .body = body,
        .header = header
    };

    print_payload(&payload);
    return 0;
}
```

```
void print_payload(Payload *payload) {
    printf("Payload %p\n", payload);
    printf("Header:\n");
    printf("- Name: %s\n", payload->header.name);
    printf("Body:\n");

    for(int i = 0; i < payload->body.num_elements; i++) {
        printf("- Content[%d]: %s\n", i, payload->body.contents[i].text);
    }
}
```

Die Funktion `print_payload` übernimmt einen Pointer auf die `payload`-Struct und gibt die gespeicherten Werte aus.

Die Variable `num_elements` ermöglicht es, nur soviele Fließtexte (`contents`) auszugeben, wie auch gespeichert wurden.

- **Structs** (strukturierte Datentypen) sind eine Möglichkeit, sich aus verschiedenen Datentypen eine eigene Variable zusammenzusetzen
- Das ist immer dann sinnvoll, wenn verschiedene Variablen **semantisch** (also vom Sinn her) zusammenhängen. Beispiel: Koordinaten **x** und **y**
- Werden Variablen in einer **Struct** zusammengefasst vereinfacht das auch die **nachträgliche Änderung** und **Refakturierung** (also die manuelle oder automatisierte Strukturverbesserung von Quelltexten unter Beibehaltung des beobachtbaren Programmverhaltens). Beispiel: die Koordinaten **x** und **y** sollen um eine dritte Dimension **z** erweitert werden
- Mehrere **Structs** desselben Typs können mittels **Array** angelegt werden
- **Structs** können sowohl per **Call-by-Reference** als auch per **Call-by-Value** an eine Funktion übergeben werden
- Auf die einzelnen Strukturvariablen wird mittels **Punktoperator** bzw. mittels **Pfeiloperator** (Pointer) zugegriffen

```
typedef struct typeName {  
    datentyp var1;  
    datentyp var2;  
    ...  
} alias;  
  
int main(){  
    alias structName;  
    ...}
```

# Weitere spezielle Datentypen

## Enum, Unions

*enum* ist ein Schlüsselwort, das zur Aufzählung von Konstanten verwendet wird und dadurch zur Strukturierung des Codes beiträgt

- Syntax:

```
enum name {const1, const2};
```

entspricht der 0

- Beispiel

```
enum Week {monday, tuesday, wednesday, thursday, friday, saturday, sunday};
```

entspricht der 6

- Deklaration und Initialisierung im Code

```
enum Week day = wednesday;
```

- Enum-Werte haben per default einen Abstand von +1, angefangen bei 0
- Wird einem Enum-Wert (oder mehreren) ein anderer Wert zugewiesen, so folgen die darauffolgenden Zahlen entsprechend

- Syntax:

```
enum name {const1, const2};
```

- Beispiel

```
enum Week {monday = 1, tuesday, wednesday, thursday, friday, saturday, sunday};
```

entspricht der 1



- Deklaration und Initialisierung im Code

```
enum Week day = wednesday;
```

entspricht der 7



Natürlich können Enumerationsen auch mit typedef in ihrer Notation vereinfacht werden


- Syntax:

```
typedef enum {const1, const2} alias;
```

- Beispiel

```
typedef enum {monday, tuesday, wednesday, thursday, friday, saturday, sunday} Week;
```

bei typedef kommt der  
Alias hinter der Deklaration



- Deklaration und Initialisierung im Code

```
Week day = wednesday;
```

- Enumerationsen können einerseits als Enum-Variable deklariert und verwendet werden, wie oben gezeigt
- Oder die Enum-Werte werden direkt im Code verwendet

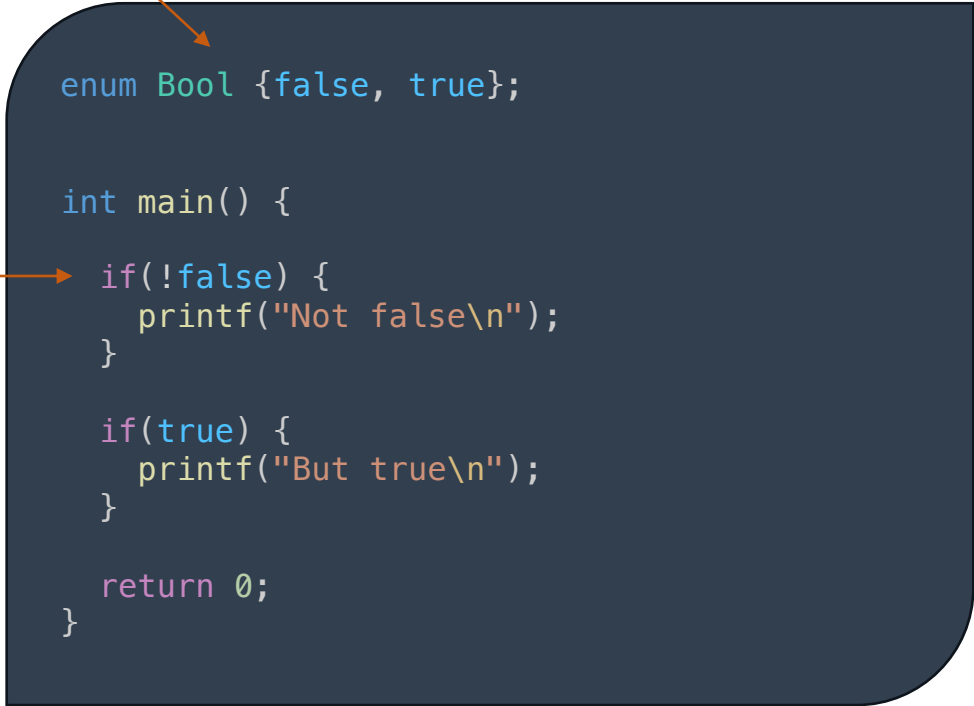


# Enumeration

## Verwendung von Enum-Werten im Code

Deklaration einer `enum` namens `Bool` mit den Werten `false` (`=0`) und `true` (`=1`)

Die `enum`-Werte werden direkt im Code verwendet und dienen hier als Stellvertreter für die jeweiligen Zahlenwerte



```
enum Bool {false, true};

int main() {
    if(!false) {
        printf("Not false\n");
    }

    if(true) {
        printf("But true\n");
    }

    return 0;
}
```

Das gleiche Ergebnis lässt sich mittels `#define` erreichen:

```
#define false 0
#define true 1
```

Unions sind eine Sonderform der Structs und eine speicherplatzsparende Möglichkeit, Daten zu strukturieren

- Von der Syntax her sind Structs und Unions gleich (bis auf den Bezeichner)
- Unions verbrauchen jedoch immer nur so viel Speicherplatz, wie die größte Union-Variable (im Beispiel rechts also den einer Double-Variable)
- Allerdings kann auch immer nur eine Union-Variable verwendet werden, im Beispiel entweder die Integer-Variable oder die Double-Variable. Die Union-Variablen sind also exklusiv verwendbar
- Das ist praktisch wenn der Speicherplatz knapp ist oder wenn das Verhalten, dass eine Variable exklusiv verwendet wird, sinnvoll ist (nested unions mit structs)

```
typedef union typeName {  
    int var1;  
    double var2;  
  
    ...  
} alias;  
  
int main(){  
    alias unionName;  
    ...}
```

- Structs, Unions und Aufzählungen (enums) dienen dazu den Code lesbarer zu machen und besser zu strukturieren
  - Mit Structs können eigene, kombinierte Datentypen angelegt werden. Das ist immer dann praktisch, wenn Variablen semantisch zusammengehören aber keine Liste sind (dann passen Arrays besser)
  - Unions eignen sich bei geringem Speicherplatz oder wenn die Unionvariablen exklusiv verwendet werden sollen
  - Enumerationen machen Aufzählungen lesbarer
- Typedef ist eine Möglichkeit, Datentypen einen eigenen Bezeichner zu geben
  - Sinnvoll bei komplexen Datentypen wie Structs oder Enumerationen