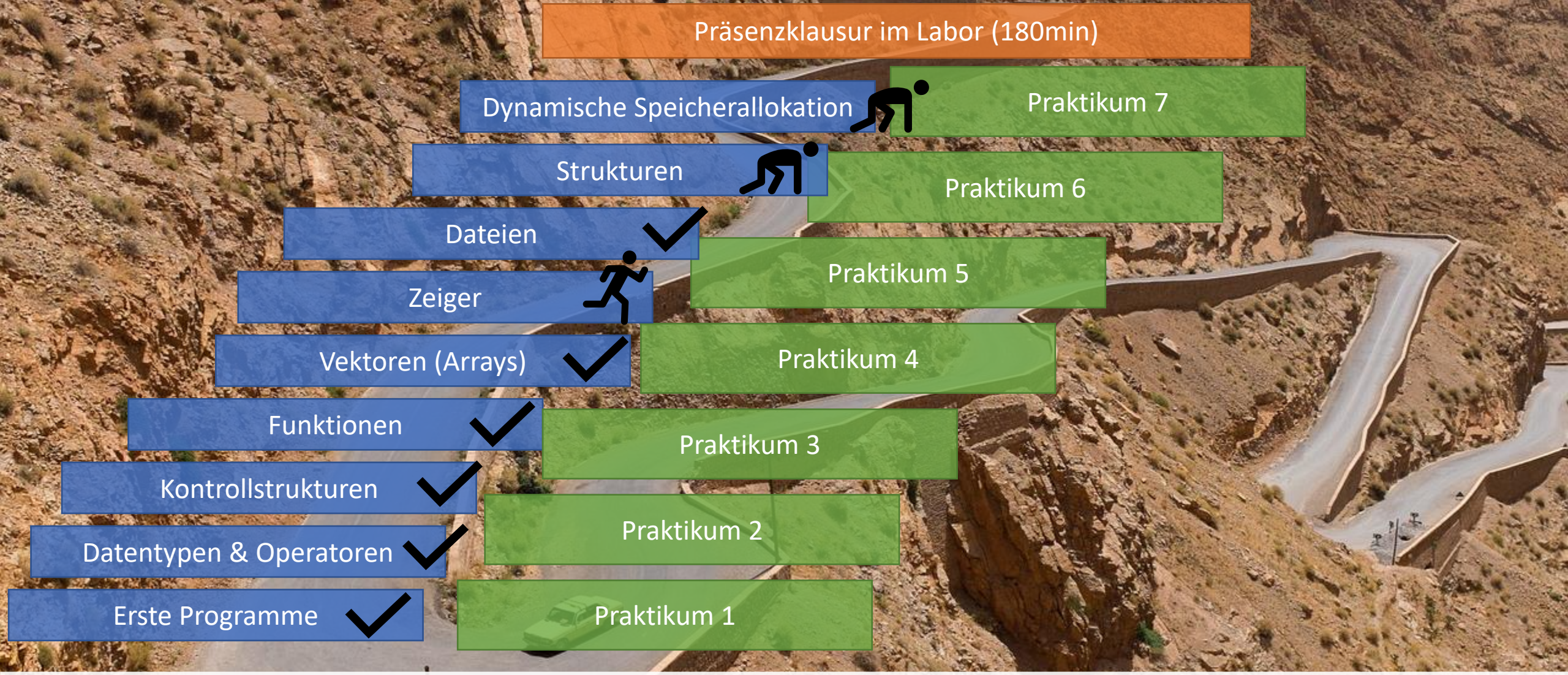
PROGRAMMIEREN I

WS 2022

Prof. Dr.-Ing. Kolja Eger
Hochschule für Angewandte Wissenschaften Hamburg

HAW HAMBURG



Unser Weg durch das Semester

Was machen wir heute?

Dynamische Speicherallokation

- Motivation
- Notwendige Schritte
- Funktion in C
  - `malloc()`
  - `calloc()`
  - `realloc()`

Spez. Datentypen

- Strukturen

DYNAMISCHE SPEICHERVERWALTUNG

HAW HAMBURG

Es soll ein Telefonbuch programmiert werden.

Wie können Sie beliebig viele Einträge realisieren (z.B. sowohl 100 Einträge als auch 1,000,000)?

Speicherverwaltung

Statisch

- Bei der Definition von Variablen wird im Hintergrund automatisch Speicher allokiert
- Bei Vektoren wird die Anzahl der Elemente festgelegt
- D.h. Speicherbedarf wird beim Programmieren ermittelt/abgeschätzt und fest „einprogrammiert“

Dynamisch

- Speicher wird zur Laufzeit des Programms allokiert, d.h. zugewiesen
- D.h. die Größe des Speichers wird dynamisch angepasst
- Hierfür wird ein Zeiger genutzt, welcher auf ein Speicherblock zeigt
- Reservierung und Freigabe des Speichers sind jetzt Aufgabe der Programmierer*innen

- Es ist nicht immer möglich den Speicherbedarf vorab abzuschätzen
- Es ist auch nicht effizient zu viel Speicher vorab zu reservieren

Speicherreservierung mit `malloc()`

1. Speicher reservieren

2. Speicher verwenden

3. Speicher freigeben

Speicherreservierung mit `malloc()`

1. Speicher reservieren
2. Speicher verwenden
3. Speicher freigeben

- Funktion `malloc()` reserviert den Speicher
- Als Parameter wird die Größe des geforderten Speichers in Bytes übergeben
- Rückgabewert ist ein Zeiger auf den Speicherblock oder `NULL` im Fehlerfall
- Rückgabewert ist vom Typ `void*` , da unbekannt für was der Speicher genutzt werden soll
- Konvertierung auf den richtigen Datentyp z.B. mit `(int*)` empfohlen (aber nicht zwingend in C)
- Speicher wird nur reserviert und nicht initialisiert. Im Speicher stehen beliebige/undefinierte Daten (→ vgl. nächstes Beispiel mit `calloc`)

Speicherreservierung mit `malloc()`

1. Speicher reservieren
2. Speicher verwenden
3. Speicher freigeben

- War die Speicherreservierung erfolgreich kann mit dem Speicher gearbeitet werden
- Zugriff über den Zeiger, in dem die Adresse des reservierten Speichers gespeichert wurde
- Fehlerbehandlung! Prüfen Sie ab, ob der Zeiger nicht `NULL` ist
- Beim Zugriff kann auch die Syntax für Vektoren genutzt werden (Verwandtschaft Zeiger & Vektoren!) und mit einem Index auf die Daten zugegriffen werden
- Man muss in dem reservierten Speicherbereich bleiben da es ansonsten zu Laufzeitfehlern kommt (Segmentation Fault!)
- Verantwortung in den Speichergrenzen zu bleiben, liegt bei der/dem Programmierer*in

Speicherreservierung mit `malloc()`

1. Speicher reservieren
2. Speicher verwenden
3. Speicher freigeben

- Wenn der Speicher nicht mehr gebraucht wird, muss er freigegeben werden
- Funktion `free()` gibt den Speicher frei und benötigt einen Zeiger auf den Speicherblock als Parameter
- Typischer Fehler ist es die Freigabe zu vergessen. Verfügbarer Speicher nimmt fortlaufend ab bis kein Speicher im System mehr zur Verfügung steht → *Memory leakage*

→ Visual Studio

```c
#include <stdio.h>

int main()
{
     int* Noten = NULL;/* Zeiger für reservierten Speicher */
     int i;/* lokale Laufvariable */

     /* 1. Schritt: Speicher reservieren */
     Noten = (int*)malloc(6 * sizeof(int));
     if (Noten == NULL) printf("Nicht genug Speicher vorhanden\n");

     /* 2. Schritt: Speicher verwenden */
     if (Noten) {
          for (i = 0; i < 6; i++) {
               Noten[i] = 0;
          }
          /* ... */
     }

     /* 3. Schritt: Speicher freigeben */
     if (Noten) free(Noten);

     /* Schlussmeldung */
     if (Noten) printf("Programm erfolgreich beendet.\n");

     return 0;
}
```

Funktionen für die dynamische Speicherreservierung

- Für die dynamische Speicherreservierung gibt es neben
  - `malloc` – Speicherblock allokieren (nicht initialisiert)

weitere Funktionen

  - `calloc` – Allokiert Speicher für ein Array mit Anzahl von Elementen und initialisiert die Werte auf null
  - `realloc` – Verändert die Größe des bereits allokierten Speicherblocks auf eine neue Größe

- Freigabe erfolgt immer mit
  - `free` – Deallokiert den Speicherblock und gibt ihn frei (der Wert des übergebenen Zeigers bleibt unverändert und zeigt auf ein nicht gültigen Speicherbereich)

- Funktionen enthalten in der Bibliothek `stdlib`

Speicherreservierung mit `calloc()`

- Funktionsaufruf mit `calloc` erwartet die Anzahl der Elemente als Parameter sowie die Größe eines Elements
- Speicher wird mit null initialisiert
- Für Ganzzahlen und Gleitkommazahlen haben alle Elemente den Wert null
- Bei Zeichenketten werden alle Elemente auf null gesetzt.
  - Länge ist auch null.
  - Aber Speicher ist für alle Elemente reserviert!
  - Speicher für Terminierung mit `'\0'` nicht vergessen!

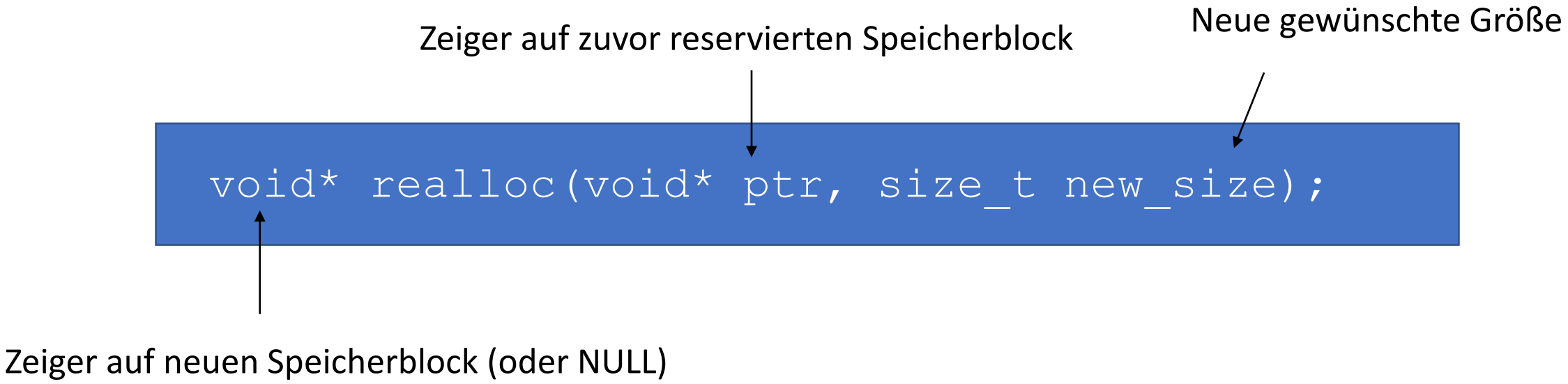
Speicherreservierung mit `calloc()`

→ Visual Studio

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

int main()
{
    int* Noten = NULL;/* Zeiger für reservierten Speicher */
    int Anzahl = 6;
    int i;/* lokale Laufvariable */

    /* 1. Schritt: Speicher reservieren */
    Noten = (int*)calloc(Anzahl, sizeof(int));
    if (Noten == NULL) printf("Nicht genug Speicher vorhanden\n");

    /* 2. Schritt: Speicher verwenden */
    if (Noten) {
        for (i = 0; i < Anzahl; i++) {
            printf("%d mal Note %d\n", Noten[i], i + 1);
        }
        /* ... */
    }
    /* 3. Schritt: Speicher freigeben */
    if (Noten) free(Noten);

    /* Schlussmeldung */
    if (Noten) printf("Programm erfolgreich beendet.\n");

    return 0;
}
```

Speichergröße ändern mit `realloc()`

- Speichergrößen können auch angepasst werden
- Hierfür wird die Funktion `realloc()` verwendet



Speichergröße ändern mit `realloc()`

- `realloc()` versucht den aktuellen Speicher zu erweitern
- Ist dies nicht möglich, wird eine neue Speicherstelle gesucht
  - Alter Speicherinhalt wird in neuen Speicherblock kopiert
  - Zeigerwert ändert sich!
- Im Fehlerfall wird NULL zurückgegeben UND der alte Speicherblock NICHT freigegeben
  - Alter Speicherblock kann im Fehlerfall weiter verwendet werden und muss vom Programmierer separat freigegeben werden
  - Um diesen Fehlerfall abzufangen und eine Speicherleckage (m*emory leakage)* zu vermeiden, braucht man zwei Zeigervariablen, um den Rückgabewert von realloc() zu überprüfen ohne den Zeiger auf den alten Speicherblock zu verlieren

```c
int main()
{
     char* text = NULL; /* Zeiger für reservierten Speicher */
     char* tmp;         /* Zeiger zur Erweiterung des Speichers */

     /* Speicher für 9 Buchstaben reservieren */
     text = (char*)malloc(10);
     if (text == NULL) printf("Nicht genug Speicher vorhanden\n");

     /* reservierten Speicher verwenden */
     if (text) {
          strcpy(text, "Guten Tag");
          printf("1. Text: %s\n", text);
     }

     /* Speicher für 33 Buchstaben reservieren */
     if (text) {
          tmp = (char*)realloc(text, 34);
          if (tmp == NULL) {
               free(text);
               printf("Nicht genug Speicher vorhanden\n");
          }
          text = tmp;
     }

     /* reservierten Speicher verwenden */
     if (text) {
          strcat(text, " meine Damen und Herren!");
          printf("2. Text: %s\n", text);
     }

     /* Speicher freigeben */
     if (text) free(text);

     return 0;
}
```

→ Visual Studio

SPEZIELLE DATENTYPEN

→ Strukturen
→ Aufzählungen
→ Union (wird in PR1 nicht im Detail behandelt!)
→ Eigene Datentypen

HAW HAMBURG

Struktur

- Fasst mehrere einfache Datentypen in einer Struktur zusammen
- Es entsteht ein komplexer Datentyp, der im Kontext des Programms eine Bedeutung hat

- Beispiele:
  - Adresse – Straße, PLZ, Ort
  - Datum – Tag, Monat, Jahr

Deklaration einer Struktur mit `struct`

Beispiel: Datum mit den Elementen Tag/Monat/Jahr

```
struct sDatum {
        int Tag;
        char Monat[10];
        int Jahr;
    };
```

- Schlüsselwort `struct` deklariert eine Struktur
- Gefolgt von einem Bezeichner/Namen für die Struktur (hier: `sDatum`)
- Liste der Elemente stehen in geschweiften Klammern
- Einzelne Elemente werden wie Variablen definiert
- Deklaration wird mit Semikolon abgeschlossen
- Deklaration legt noch keine Variable von diesem Typ an, sondern macht Struktur dem Compiler „nur“ bekannt

Definition einer Struktur

```
struct sDatum Datum;
```

Ohne oder mit Initialisierung

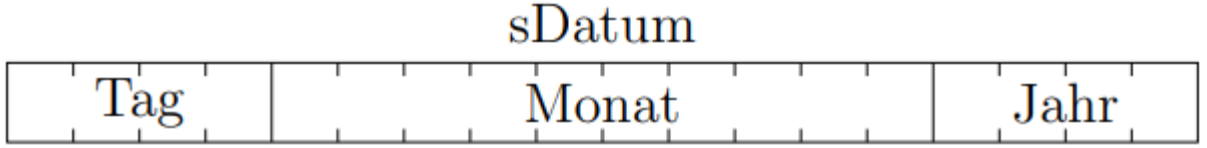
```
struct sDatum Datum = {9, "November", 2021};
```

Initialisierung ähnlich wie bei Vektoren mit {}

Struktur im Speicher

- Struktur liegt nacheinander angeordnet im Speicher
- Evtl. kleinere Lücken zwischen den Variablen, da Compiler häufig auf ein festgelegtes Vielfaches von Bytes den Speicher ausrichtet
- Tatsächliche Größe der Struktur kann mit `sizeof` ermittelt werden

Anordnung der Struktur sDatum im Speicher



→ Visual Studio

```c
#include <stdio.h>

// erste Struktur deklarieren
struct sDatum {
    int Tag;
    char Monat[10];
    int Jahr;
};

void main() {
    // Variable von neuer Struktur definieren
    struct sDatum Datum = { 9, "November", 2021 };

    // Ausgabe struct -> structName.ElementName
    printf("Datum: %d.%s %d\n", Datum.Tag, Datum.Monat, Datum.Jahr);

    // Werte können auch später zugewiesen werden
    struct sDatum Datum2;

    Datum2.Tag = 9;
    strcpy(Datum2.Monat, "November");
    Datum2.Jahr = 2021;

    printf("Datum(2): %d.%s %d\n", Datum2.Tag, Datum2.Monat, Datum2.Jahr);

    …
}
```

VIELEN DANK FÜR IHRE AUFMERKSAMKEIT!

HAW HAMBURG