Imperative Programmierung

Übung 8: Structs und Dynamischer Speicher

Justin Kreikemeyer
Informatik, Uni Rostock

Fragen?

Leitfragen

- Wie reserviere ich meinen eigenen Speicher?
- Wie strukturiere ich meinen eigenen Speicher?
- Wofür braucht man das?

Motivation

- Ein Studierender ist ein Objekt mit einem *Namen*, einem *Alter*, einem *Geschlecht* und einer *Matrikelnummer*. Repräsentieren Sie einen Studierenden in C!
- $\rightarrow$ Aye!

```c
#include <stdio.h>
int main(int argc, char* argv[]) {
  char student_name[] = "Alfred";
  unsigned short student_age = 22;
  char student_gender = 'm';
  long student_nr = 123456789;
  return 0;
}
```

Motivation

- Ein Studierender ist ein Objekt mit einem *Namen*, einem *Alter*, einem *Geschlecht* und einer *Matrikelnummer*. Repräsentieren Sie **10** Studierende in C!
- $\rightarrow$ Uff *$\sim$*

```c
#include <stdio.h>
int main(int argc, char* argv[]) {
  // student1
  char student1_name[] = "Alfred";
  unsigned short student1_age = 22;
  char student1_gender = 'm';
  long student1_nr = 123456789;
  // student2
  char student2_name[] = "Berta";
  unsigned short student2_age = 21;
  // ...
  return 0;
}
```

Motivation

- Ein Studierender ist ein Objekt mit einem *Namen*, einem *Alter*, einem *Geschlecht* und einer *Matrikelnummer*. Repräsentieren Sie **10** Studierende in C!
- $\rightarrow$ Besser, aber immer noch uff *$\sim$*

```c
#include <stdio.h>
int main(int argc, char* argv[]) {
  // student names
  char* student_names[] = {"Alfred", "Berta", /*...*/ };
  usigned short student_ages[] = {22, 21, /*...*/ };
  /*
   * btw.
   * Mehrzeiliger Kommentar geht so
   */
  // ...
  return 0;
}
```

Motivation

- Ein Studierender ist ein Objekt mit einem *Namen*, einem *Alter*, einem *Geschlecht* und einer *Matrikelnummer*. Repräsentieren Sie **10** Studierende in C!
- $\rightarrow$ Wir legen einfach unseren eigenen Datentypen an, der mehrere (heterogene) elementare Datentypen gruppiert!

```c
#include <stdio.h>
typedef struct _student {
  char name[];
  usigned short age;
  char gender;
  long nr;
} student;
int main(int argc, char* argv[]) {
  // students
  student my_students[10];
  return 0;
}
```

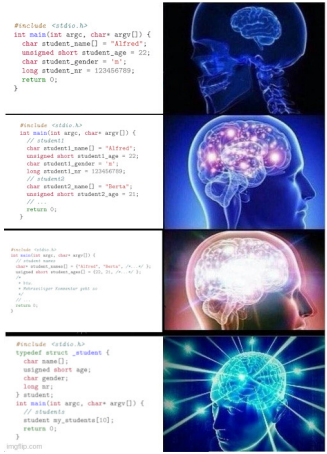
Motivation: Zusammenfassend...

(Variable für jedes Attribut)

Variable für jedes Attr. für jeden Studierenden

Ein Array für jedes Attribut

Eigene Struktur und Typ



Structs in C

- Definition des Formates eines Speicherbereichs (≈ Typ)
- `"struct" <name> "{" <decls> "};"`
- `<decls> = <typ> <name> ";" { <typ> <name> ";" }`
- Deklaration von Variablen mit `struct my_struct name;`
- Inline-Initialisierung mittels `{...}` möglich (wie Array); nun heterogen!

```
struct my_struct {
  int x;
  char* s;
};
// ...
struct my_struct test = { 42, "Die Antwort auf alles." };
```

Operationen mit Structs

- Zugriff auf einzelne Felder mittels `.`-Operator, z.B. `s.feld = 10`
- Falls Zeiger-Typ, dann Abkürzung `(*s).feld` mit `s->feld`

```
struct my_struct {
  int x;
  char* s;
};                              //          111111111122
// ...                           012345678901234567890
struct my_struct test = { 42, "Die Antwort auf alles." };
struct my_struct* test_ptr = &test;
test.x = 12;
test_ptr->s[21] = '!';
printf("%d", test.x); // Ausgabe?
printf("%s", test.s); // Ausgabe?
```

Eigener Typ mittels Typedef

- Syntax: "typedef" <typ> <name> ";"

```
struct _my_struct {
  int x;
  char* s;
};
typedef struct _my_struct my_struct;
// ODER gleich
typedef struct _my_struct {
  int x;
  char* s;
} my_struct;
// -> my_struct steht nun für den Typen "struct _my_struct"
// Auch hilfreich
typedef struct my_struct* my_struct_ptr;
// -> my_struct steht nun für den (Zeiger-)Typen "struct _my_struct*"
```

Gemeinsames Beispiel

Definieren Sie eine Struktur Auto, welche die Farbe, maximale Geschwindigkeit und Kilometerstand speichert, sowie ob es sich dabei um einen Sportwagen handelt. Es gibt nur die drei Lackierungen rot, grün und blau.

Motivation: Dynamischer Speicher

- Schreiben Sie eine Funktion, die zur besseren Übersicht einen neuen Studierenden mit entsprechenden Attributen anlegt und zurück gibt!

```c
#include <stdio.h>
typedef struct _student {/*...*/} student;
student* make_student(char* name, unsigned short age, /*...*/) {
  student temp;
  temp.name = name;
  // ...
  return &temp;
}
int main(int argc, char* argv[]) {
  student* my_student = make_student("Alfred", 22, /*...*/);
  return 0;
}
```

Was passiert, wenn wir das ausführen?

Konzept: Dynamischer Speicher

- Programm unterteilt in verschiedene Speicherbereiche
  - Stack $\approx$ “von C” verwalteter Speicherbereich für lokale (in Funktion, Block definierte) Variablen
  - Heap $\approx$ nicht verwalteter Speicherbereich für globale Variablen
  - (und noch ein paar mehr für Variablen, Funktionen, etc.; s. Vorlesung)
- Standard-Speicherort: Stack; **Lebensende automatisch am Ende eines Blocks**
- Für volle Kontrolle: Heap; Speicher muss **vom Programmierer** verwaltet (z.B. freigegeben) werden! Fehleranfällig, da z.B. illegaler Zugriff möglich

Dynamischer Speicher in C

- Funktion `malloc(<size>)` zum allozieren eines Speicherbereichs der Größe `<size>` auf dem Heap
- Weitere, wie `realloc()` zum vergrößern etc. (s. Dokumentation)

```c
#include <stdio.h>
#include <stdlib.h>  // für malloc, calloc, realloc, ...
typedef struct _student { /*...*/ } student;
student* make_student(char* name, unsigned short age, /*...*/) {
  student* temp = malloc(sizeof(student));
  temp->name = name;
  // ...
  return temp;
}
int main(int argc, char* argv[]) {
  student* my_student = make_student("Alfred", 22, /*...*/);
  student* my_students = malloc(2 * sizeof(student));
  my_students[0] = my_student;
  return 0;
}
```

Dynamischer Speicher in C

- Funktion `malloc(<size>)` zum allozieren eines Speicherbereichs der Größe `<size>` auf dem Heap
- Weitere, wie `realloc()` zum vergrößern etc. (s. Dokumentation)

```c
#include <stdio.h>
#include <stdlib.h>  // für malloc, calloc, realloc, ...
typedef struct _student { /*...*/ }* student;
student make_student(char* name, unsigned short age, /*...*/) {
  student temp = malloc(sizeof(struct _student));
  temp->name = name;
  // ...
  return temp;
}
int main(int argc, char* argv[]) {
  student my_student = make_student("Alfred", 22, /*...*/);
  student my_students = malloc(2 * sizeof(struct _student));
  my_students[0] = my_student;
  return 0;
}
```

NULL

- Nicht initialisierter Zeiger zeigt immer auf die Adresse 0 (in C `NULL`)
- Beim versuchen auf diese Adresse zuzugreifen, bekommt man (bestenfalls) einen Fehler
- Falls bei `malloc` ein Problem auftritt (z.B. zu wenig Speicher frei), dann wird auch `NULL` zurück gegeben
- $\rightarrow$ Fehlerfall behandeln! (privat empfohlen; Klausur/HA Pflicht)

```c
#include <stdio.h>
#include <stdlib.h>  // für malloc, calloc, realloc, ...
int main(int argc, char* argv[]) {
  int* arr = malloc(10 * sizeof(int));
  if (arr == NULL) {
    printf("Fehler beim Reservieren von Speicher!\n");
    return 1;    // <- Hier nun ein Fall, wo return 1 in main sinnvoll ist :)
  }
  return 0;
}
```

Speicherfreigabe

- Speicherbereich kann mit `free(<ptr>)` freigegeben werden
- Manuelles Beenden der Lebensdauer einer Variablen
- Notwendig bei lang laufenden Programmen ($\rightarrow$ “memory leak”)
- Bei kurzen Programmen: Es wird beim Beenden sowieso aufgeräumt...

```c
#include <stdio.h>
#include <stdlib.h>  // für malloc, calloc, realloc, ...
int main(int argc, char* argv[]) {
  int* arr = malloc(10 * sizeof(int));
  if (arr == NULL) {
    printf("Fehler beim Reservieren von Speicher!\n");
    return 1;    // <- Hier nun ein Fall, wo return 1 in main sinnvoll ist :)
  }
  // ...
  free(arr);
  arr[0] = 10; // Fehler: Pointer used after free
  return 0;
}
```

Gemeinsames Beispiel

Schreiben Sie eine Funktion, welche eine Matrix mit den (variablen) Dimensionen $n \times m$ erzeugt und zurück gibt.

Fragen?

Aufgaben: Structs und Dynamischer Speicher

Schreiben Sie die folgenden C-Programme und Funktionen[1]! Nutzen Sie dazu die Konzepte aus dieser Übung und das Cheat Sheet!
Bearbeitungszeit: bis 10 Minuten vor Schluss. Dann Besprechung von häufigen Problemen.

[1] Weitere Aufgaben können jederzeit beim Übungsleiter erfragt werden.

Strukturen: Punkte

Scalar Definieren Sie den Typen `Scalar`, der eine (skalare) reelle Zahl repräsentiert mittels `typedef`.

Vector2D Definieren Sie den Typen `Vector2D`, der einen zwei-dimensionalen Vektor mit den Komponenten x und y repräsentiert.

create Implementieren Sie eine Funktion `create_vector`, die einen neuen (Null-)Vektor erstellt und zurück gibt. Die Werte der Komponenten sollen als Parameter übergeben werden.

scale Implementieren Sie eine Funktion `scale`, welche einen Skalar und einen Vektor übergeben bekommt und den Vektor entsprechend skaliert. Das Ergebnis ist ein neuer (skalierter) Vektor.

add Implementieren Sie eine Funktion `add`, welche zwei Vektoren übergeben bekommt und ihre Summe berechnet und zurück gibt.

product Implementieren Sie eine Funktion `product`, welche das Skalarproduct zweier Vektoren berechnet.

Strukturen: Punkte

VectorND Erweitern Sie die Struktur `Vector` so, dass sie eine Variable Anzahl n an Komponenten (sowie n selbst) aufnehmen kann.

IncreaseDim Implementieren Sie eine Funktion `struct vectorNd* increase_dim(struct vectorNd* vec, int new_dims`, welche die Anzahl an Komponenten eines Vektors erhöht. Achten Sie auf sinnvolle Tests auf mögliche Werte von `new_dims`. Die bestehenden Werte sollen als erste Komponenten in den neuen, größeren Vektor übertragen werden.