Dynamische Felder

Allgemeine Anmerkungen zur Aufgabenstellung:

- Beantworten Sie die Kontrollfragen in `TUWEL` bevor Sie mit der Übung beginnen.
- Legen Sie für `Teil A` der Angabe ein neues `Code::Blocks`-Projekt an und erweitern Sie dieses schrittweise für die darauffolgenden Punkte.
- Stellen Sie während der Übung laufend sicher, dass Ihr Programm kompilierbar ist und richtig funktioniert.
- Die Abgabe erfolgt durch Hochladen Ihres vollständigen `Code::Blocks`-Projektordners als zip-Datei (`Matrikel-Nr_UE1.zip`) in `TUWEL`. Anschließend melden Sie sich für das Abgabegespräch in `TUWEL` an. Dieses erfolgt über die Telekonferenz-App `Zoom` (www.zoom.us). Stellen Sie sicher, dass diese App rechtzeitig vor Beginn des Abgabegespräches auf Ihrem PC installiert ist und dass Sie Ihr aktuelles Übungsprogramm in `Code::Blocks` für die Abgabe geöffnet haben. Testen Sie bitte auch die korrekte Funktionalität Ihrer Videokamera, da Sie zu Beginn Ihres Abgabegespräches Ihren Studentenausweis herzeigen müssen und auch während des Gesprächs eine aktive Videoverbindung erforderlich ist.
- Beachten Sie, dass Upload und Terminauswahl für das Abgabegespräch erst möglich sind, wenn Sie 80% der Kontrollfragen richtig beantwortet haben. Bis **19.10.2021 23:59** müssen diese Fragen positiv beantwortet, Ihre Ausarbeitung hochgeladen und ein Termin für das Abgabegespräch ausgewählt werden.
- Inhaltliche Fragen stellen Sie bitte in der Fragestunde oder im zugehörigen Forum in `TUWEL`.
- Für eine positive Beurteilung Ihrer Abgabe **muss** diese kompilierbar sein, Ihr Programm fehlerfrei terminieren und die in der Angabe angeführten Funktionen mit den vorgegebenen Funktionsköpfen enthalten. Des Weiteren müssen die Funktionen hinreichend getestet sein. Sie müssen in der Lage sein, Ihr Programm beim Abgabegespräch zu erklären!

Folgende Hilfsmittel stehen Ihnen in `TUWEL` zur Verfügung:

- Alle bisherigen Referenzbeispiele sowie Vorlesungsfolien
- Interaktive Jupyter-Notebooks auf `prog1.iue.tuwien.ac.at`
- Das Buch zur LVA "Programmieren in C" (Robert Klima, Siegfried Selberherr)
- Kurzanleitung für die Entwicklungsumgebung `Code::Blocks`

Hinweis: Es wird erwartet, dass alle Abgaben zu den Übungen **eigenständig** erarbeitet werden und wir weisen Sie darauf hin, dass alle Abgaben in einem standardisierten Verfahren auf Plagiate überprüft werden. Bei Plagiatsvorfällen entfällt das Abgabegespräch und es werden keine Punkte für die entsprechende Abgabe vergeben.

Stoffgebiet
Sie benötigen für diese Übung zusätzlich zum gesamten Lehrinhalt der Lehrveranstaltung Programmieren 1 folgenden Stoff:

- Zeiger (Kapitel 14)
- Speicher reservieren und freigeben (Kapitel 20)

Einleitung
Für das Speichern und Manipulieren von Datenreihen werden häufig Felder verwendet. In Programmieren 1 mussten Sie die Größe eines Feldes bereits vor dem Kompilieren festlegen. Diese konnte während der Ausführung des Programms nicht mehr verändert werden. Dies stellt eine weitreichende Einschränkung der Verwendbarkeit auf Datensätze unbekannter Größe dar. In dieser Übung lernen Sie, Felder dynamisch zur Laufzeit zu erstellen, deren Größe zu verändern, und den angeforderten Speicherbereich nach der Benutzung wieder freizugeben.

Insbesondere in Computersimulationen werden Felder zur Repräsentation von Vektoren und Matrizen verwendet. Hier werden Sie dynamische Felder verwenden, um Vektoren in der Programmiersprache C darzustellen. Weiters werden Sie verschiedene Operationen für diese Vektoren implementieren. Achten Sie dabei auf eine korrekte Speicherverwaltung!

DUMA Bibliothek
Die unsachgemäße Verwendung von dynamischen Speicherbereichen kann zu schwer auffindbaren Programmfehlern zur Laufzeit führen. Um die Fehlersuche zu vereinfachen und die korrekte dynamische Speicherverwaltung sicherzustellen, verwenden Sie die `DUMA` Bibliothek. Mit Hilfe dieser Bibliothek bekommen Sie eine Rückmeldung über die Verwendung von dynamisch alloziertem Speicher. Diese Bibliothek ersetzt die Systemfunktionen `malloc()`, `calloc()`, `realloc()` und `free()` durch eigene Varianten, die bei falscher Verwendung unmittelbar eine Fehlermeldung zurück liefern. Für die Übung haben wir `DUMA` so verändert, dass über eine Umgebungsvariable (`DUMA_MALLOC_ERROR`, siehe unten) die Fehlerwahrscheinlichkeit für die Speicherallozierung eingestellt werden kann.

Um die `DUMA`-Bibliothek zu installieren und in Ihrem Projekt zu verwenden, folgen Sie den Anweisungen in der auf `TUWEL` zur Verfügung gestellten `DUMA`-Anleitung.

Um ein gescheitertes Anfordern von Speicherbereichen zu simulieren, starten Sie unter Linux Ihr Programm in der Konsole mit dem Befehl

```
DUMA_MALLOC_ERROR=50 ./<Programmname>
```

Unter Windows verwenden Sie `cmd` und starten Ihr Programm wie folgt

```
set DUMA_MALLOC_ERROR=50
<Programmname>.exe
```

Die Umgebungsvariable `DUMA_MALLOC_ERROR` gibt dabei die Fehlerwahrscheinlichkeit in Prozent an. In diesem Beispiel würde also im Mittel jede zweite Speicheranforderung fehlschlagen.

Hinweis: Beachten Sie, dass während des Abgabegesprächs auch die korrekte Verwendung der dynamischen Speicherverwaltung überprüft wird. Achten Sie daher u.a. auf Speicherlecks und korrekte Freigaben. Ihr Programm muss bei einer gescheiterten Speicher-Allozierung eine definierte Fehlerbehandlung durchführen und darf unter **keinen Umständen** abstürzen!

Teil A

✘ Fügen Sie die `DUMA` Bibliothek zu Ihrem Projekt hinzu bevor Sie mit der eigentlichen Implementierung beginnen. Die genauen Anweisungen zur Installation und Verwendung dieser Bibliothek finden Sie in der zugehörigen `DUMA`-Anleitung auf `TUWEL`.

✘ Definieren Sie einen Aufzählungstyp `error_t`, der verschiedene Fehlercodes bereitstellt.

```
typedef enum error_e
{
  SUCCESS=0,
  NULL_POINTER_ERROR,
  ALLOC_FAILURE_ERROR,
  OUT_OF_RANGE_ERROR,
  DIMENSION_MISMATCH_ERROR
} error_t;
```

Die von Ihnen in dieser Übung implementierten Funktionen sollen einen entsprechenden Fehlercode zurückliefern, falls ein Fehler auftritt.

✘ Definieren Sie einen Verbunddatentyp `vector_t`, der einen Vektor mit der Dimension `dim` repräsentiert und dessen Elemente im **dynamischen Feld** `values` gespeichert werden.

```
typedef struct vector_s
{
  long *values;
  long dim;
} vector_t;
```

✘ Implementieren Sie die Funktion

```
error_t vector_allocate(vector_t *vec, long dim);
```

welche einen übergebenen Vektor `vec` initialisiert. Das bedeutet, seine Dimension ist mit `dim` festzulegen, Speicher für seine Elemente anzufordern sowie diese mit `0` zu initialisieren. Fangen Sie mögliche Fehler ab und geben Sie einen entsprechenden Fehlercode vom Typ `error_t` zurück.

Hinweis: Beachten Sie hierbei, dass die Adresse auf den übergebenen Vektor `vec` von einer **statisch** im Hauptprogram angelegten Variable sein muss, also z.B.

```
vector_t vec = { NULL, 0 };
vector_allocate(&vec, 10);
```

Überlegen Sie, warum folgender Code nicht funktionieren würde

```
vector_t *vec = NULL;
vector_allocate(vec, 10);
```

Wir empfehlen Ihnen weiters, eine Variable vom Typ `vector_t` immer mit

```
vector_t vec = { NULL, 0 };
```

zu initialisieren, um erkennbar zu machen, dass noch kein dynamischer Speicher in `values` angefordert wurde.

✘ Implementieren Sie weiters die Funktion

```
error_t vector_clear(vector_t *vec);
```

welche den zuvor reservierten Speicher für die Elemente `values` wieder freigibt, den Zeiger auf `NULL` und die Dimension des Vektors auf `0` setzt.

✘ Um den Vektor mit Werten zu füllen, schreiben Sie die Funktion

```
error_t vector_fill(vector_t *vec, long index, long value);
```

welche den Wert `value` an die Stelle `index` im Vektor `vec` schreibt. Falls `index` ungültig sein sollte, geben Sie einen entsprechenden Fehlercode zurück.

✘ Um die Elemente des Vektors anzuzeigen, implementieren Sie die Funktion

```
error_t vector_print(vector_t *vec);
```

welche den Vektor `vec` in der Konsole ausgibt, beispielsweise in der Form

```
[ 3, -5, 4, 10 ]
```

✘ Erstellen Sie ein Hauptprogramm, in dem Sie mehrere Vektoren **statisch** anlegen, **dynamisch** Speicher für deren Elemente reservieren, mit Werten befüllen und ausgeben. Achten Sie insbesondere auf die korrekte Speicherfreigabe am Ende des Programmes.

Teil B

✘ Implementieren Sie die Funktion

```
error_t vector_change_dim(vector_t *vec, long new_dim);
```

welche die Dimension des Vektors `vec` auf den übergebenen Wert `new_dim` ändert. Wird die Dimension erhöht, so sollen die neuen Elemente mit 0 initialisiert werden.

Hinweis: Verwenden Sie die Funktion `realloc` und beachten Sie die im Referenzbeispiel diskutierten Feinheiten.

✘ Implementieren Sie zusätzlich die Funktion

```
error_t vector_add(vector_t *vec_a, vector_t *vec_b, vector_t *vec_sum);
```

welche die Summe der beiden Vektoren `vec_a` und `vec_b` in den Vektor `vec_sum` schreibt.

Hinweis: Falls `values` in `vec_sum` nicht `NULL` ist, gehen Sie davon aus, dass bereits Speicher für den Ergebnis-Vektor reserviert wurde. Reagieren Sie entsprechend, um eventuelle Speicherlecks zu vermeiden.

✘ Implementieren Sie die Funktion

```
error_t vector_slice(vector_t *vec, long index_a, long index_b);
```

welche das Feld `values` des Vektors `vec` auf den durch `index_a` und `index_b` gegebenen Bereich zuschneidet. Nach der Funktionsausführung soll das Feld `values` also die Elemente von `index_a` bis exklusive `index_b` vom Feld vor der Funktionsausführung beinhalten.

✘ Testen Sie die neuen Funktionen in Ihrem Hauptprogramm. Achten Sie insbesondere auf die Behandlung auftretender Fehler.

Teil C

✘ In den Teilen A und B wurden die Vektoren selbst statisch angelegt, während der Speicher für die jeweiligen Elemente dynamisch verwaltet wurde. Implementieren Sie nun eine neue Variante der Funktion `vector_allocate`, welche den gesamten Verbunddatentyp `vector_t` dynamisch alloziert, richtig initialisiert und einen Zeiger auf den neu erstellten Vektor zurückliefert.

```
vector_t *vector_allocate_dynamic(long dim);
```

✘ Implementieren Sie weiters eine Funktion

```
error_t vector_free(vector_t *vec);
```

welche den dynamisch allozierten Vektor `vec` inklusive seiner Elemente wieder freigibt. Achten Sie hierbei auf die richtige Reihenfolge der Freigaben.

✘ Modifizieren Sie Ihr bisheriges Hauptprogramm, um die voll-dynamische Allozierung zu testen. Verwenden Sie beispielsweise Präprozessor-Anweisungen um zwischen beiden Varianten umschalten zu können.

Verwenden Sie für die weitere Übung ausschließlich dynamisch allozierte Vektoren.

Teil D

✘ Definieren Sie einen neuen Verbunddatentyp `vec_array_t`, der mehrere Vektoren dynamisch speichern kann.

```
typedef struct vec_array_s
{
  vector_t **vectors;
  long length;
} vec_array_t;
```

Überlegen Sie sich, warum hier ein Doppelzeiger für das dynamische Feld von Vektoren notwendig ist.

✘ Implementieren Sie für den neuen Typ `vec_array_t` eine Funktion

```
vec_array_t *vec_array_allocate(long length);
```

welche Speicher für die Struktur `vec_array_t` sowie für das Feld `vectors` dynamisch anfordert. Die Funktion soll das neue Array richtig initialisieren und einen Zeiger darauf zurückgibt.

✘ Schreiben Sie eine Funktion

```
error_t vec_array_free(vec_array_t *array);
```

die `array` und alle darin enthaltenen Vektoren wieder freigibt.

✘ Schreiben Sie eine Funktion

```
error_t vec_array_store(vec_array_t *array, vector_t *vec);
```

welche einen übergebenen Vektor `vec` am Ende von `vectors` in `array` einfügt.

Hinweis: Vergrößern Sie dazu den Speicherbereich mit `realloc`.

✘ Implementieren Sie die Funktion

```
error_t vec_array_delete_at(vec_array_t *array, long index);
```

welche den Vektor an der Stelle `index` aus `vectors` in `array` entfernt. Achten Sie hierbei darauf, dass `index` gültig ist und keine Lücken im Speicher entstehen.

✘ Testen Sie Ihre Funktionen im Hauptprogramm, indem Sie Vektoren in ein Array einfügen und danach wieder löschen.

Teil E

✘ Implementieren Sie eine Funktion

```
error_t vec_array_save(vec_array_t *array, char *file_name);
```

welche ein übergebenes Vektor-Array in einer Textdatei speichert

✘ Schreiben Sie weiters eine Funktion

```
vec_array_t *vec_array_load(char *file_name);
```

um ein zuvor in einer Textdatei gespeichertes Array wieder zu laden. Die Funktion soll einen Zeiger auf das geladene Array zurückgeben.

Hinweis: Überlegen Sie sich zum Speichern und Laden ein geeignetes Dateiformat, um das Vektor-Array **vollständig** wiederherstellen zu können.

Teil F

✘ Definieren Sie einen Datentyp für Funktionen (Funktionszeiger), welche für einen übergebenen Vektor einen `double` Wert zurückliefert.

```
typedef double (*measure_f) (vector_t *);
```

Ein solcher Funktionszeiger kann daher beispielsweise die Norm oder das Minimum/Maximum eines Vektors repräsentieren.

✘ Implementieren Sie die Funktion

```
double vector_norm(vector_t *vec);
```

welche die euklidische Norm des Vektors `vec` berechnet und zurückgibt. Die euklidische Norm eines n-dimensionalen Vektors $\boldsymbol{v}$ ist gegeben durch:

$$\|\boldsymbol{v}\|_2 := \sqrt{(v_1)^2 + (v_2)^2 + \cdots + (v_n)^2} = \left(\sum_{i=1}^{n} (v_i)^2\right)^{1/2}$$

✘ Schreiben Sie eine Funktion

```
error_t vec_array_sort(vec_array_t *array, measure_f sort_key);
```

welche die Vektoren im übergebenen Array aufsteigend sortiert. Als Sortierschlüssel dient dabei eine Funktion, die mittels eines Funktionszeigers `sort_key` übergeben wird.

✘ Wenden Sie nun Ihre Sortierfunktion auf ein Vektor-Array an und überprüfen Sie ob diese korrekt arbeitet. Als Sortierschlüssel verwenden Sie `vector_norm`.