IFM 3.3 Systemprogrammierung, W19

Blatt 02: Segment-Anzeige und Studi-Verwaltung [ANSI-C] (20 Punkte)

Praktikum: 31.10.2019

Carsten Gips, Birgit C. George, FH Bielefeld

1 Vorgaben

	LED-Segmentanzeige	Studi-Verwaltung	Prüfungsanmeldungen
Unterordner	`ledanzeige`	`studiverwaltung`	`pruefungen`
Dateien	`segmentanzeige.[h,c]`	`studiverwaltung.[h,c]`	`pruefungen.[h,c]` `display.[h,c]`
Test-Targets	`test_segmentanzeige`	`test_studiverwaltung`	`test_pruefungen`

Hinweis: Für die Prüfungsanmeldungen brauchen Sie weder `free()` noch `malloc()`, dito für die LED-Segmentanzeige! Erst für die Studi-Verwaltung werden teilweise Inhalte von VL04 benötigt.

Erinnerung: Bitte beachten Sie die Regeln zur Bearbeitung der Praktikumsaufgaben. Insbesondere müssen Sie ab diesem Blatt jeweils ein passendes Makefile mit den Standard-Targets anlegen und Ihren Quellcode mit Doxygen dokumentieren (selbstständige Einarbeitung!).

2 Vorbereitung

2.1 Ansteuern der LED-Segmentanzeige

2.1.1 Anschluss der LED-Segmentanzeige an den Raspberry Pi

Schließen Sie das 4-adrige Kabel an die LED-Segmentanzeige an. Achten Sie dabei auf die korrekte Ausrichtung des Steckers: Dieser kann nur in einer Ausrichtung leicht in die Buchse gesteckt werden. Am Raspberry Pi schließen Sie das Kabel im Port wie in Abbildung 1 dargestellt an. Achten Sie auch hier auf die richtige Ausrichtung des Steckers!

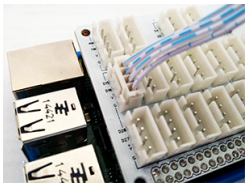

Abbildung 1: Anschluss der LED-Segmentanzeige am Raspi

2.1.2 Benennung der Pins im Code

Die Pins des genutzten Ports sind in der Vorgabe (`ledanzeige/TM1637_intern.h`) als Präprozessordirektiven (`PIN_CLOCK` und `PIN_DATA`) festgelegt. Sie können diese Literale in Ihrem Code wie Konstanten verwenden.

2.1.3 Serielle Datenübertragung an die LED-Segmentanzeige

Die Datenübertragung erfolgt bitweise seriell. Sie müssen zur Übertragung eines Bytes alle 8 Bits *einzeln* übertragen, beginnend „von rechts“, d.h. mit dem niedrigstwertigen Bit (*least significant bit*, LSB):

1. Setzen Sie den Clock-Pin auf `LOW`: Aufruf von `digitalWrite(PIN_CLOCK, LOW)`
2. Schreiben Sie das Bit auf den Daten-Pin: Aufruf von `digitalWrite(PIN_DATA, LOW)` (falls Sie den Wert 0 ausgeben wollen; für den Wert 1 ersetzen Sie `LOW` durch `HIGH`)

3. Setzen Sie den Clock-Pin auf `HIGH`: Aufruf von `digitalWrite(PIN_CLOCK, HIGH)`

Nach *jedem* Aufruf von `digitalWrite()` müssen Sie mit Hilfe von `delayMicroseconds(DELAY_TIMER)` kurz warten, damit sich die Spannung am Pin stabilisieren kann.

Nach dem Senden der 8 Bit wird die Datenübertragung mit dem Aufruf `TM1637_ack()` abgeschlossen.

Hinweis: Die genannten Symbole finden Sie in den Headern `ledanzeige/TM1637_intern.h` und `wiringPi.h`.

2.2 Studi-Verwaltung: Verkettete Listen

Auf diesem Blatt erstellen Sie Datenstrukturen und dynamische Listen zur Verwaltung von Studierendeninformationen.

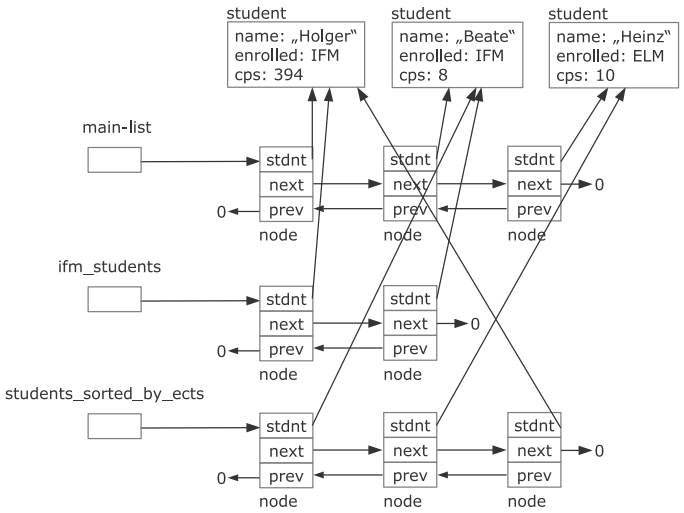

Abbildung 2: Skizze der Datenstrukturen zur Studi-Verwaltung

Die Studierenden sind in der Hauptliste `main_list` nach ihrem zeitlichen Eintreten in die FH sortiert.

Für andere Sortierungen (etwa nach Credits) oder für bestimmte Teilmengen der Studierenden (etwa alle Informatiker) werden Hilfslisten angelegt. Dabei zeigen die `stdnt`-Zeiger in den `node`-Elementen der Hilfsliste jeweils auf einen in der Hauptliste verankerten Studierenden. Dadurch werden die Studierenden nicht mehrfach angelegt/kopiert.

2.3 Prüfungsanmeldungen

Ein Dozent hat für mündliche Prüfungen eine Liste mit 10 Terminen erstellt, wo sich Studierende eintragen können.

Jeder Studierende kann sich für genau einen Termin anmelden („registrieren“). Ist der Wunschtermin bereits belegt, wird er automatisch für den nächsten freien Termin in der Liste eingetragen. Ein Studierender kann von der Prüfung zurücktreten und sich aus der Liste austragen.

Mit der LED-Anzeige kann man den Stand der Anmeldungen gut visualisieren.

3 Aufgaben

3.1 Ansteuern der LED-Segmentanzeige

3.1.1 Datentypen (`ledanzeige/segmentanzeige.h`) (1 Punkt)

1. Definieren Sie einen vorzeichenlosen Datentyp `byte`, der Werte von 0 bis 255 halten kann, also 8 Bit „breit“ ist.
2. Definieren Sie einen Aufzählungstyp `segment` für die Segmente der Anzeige mit den Elementen `SEG1` (Wert 0), `SEG2` (Wert 1), `SEG3` (Wert 2), `SEG4` (Wert 3).
3. Definieren Sie einen Aufzählungstyp `dot` für den Dezimalpunkt der Anzeige mit den Elementen `OFF` (Wert 0) und `ON` (Wert 1).

4. Definieren Sie einen Aufzählungstyp `brightness` für die Helligkeit der Anzeige mit den Elementen `DARK` (Wert 0), `MEDIUM` (Wert 1) und `BRIGHT` (Wert 7).

Thema: Umgang mit Basisdatentypen und Strukturen

3.1.2 Funktionen (`ledanzeige/segmentanzeige.c`) (2 Punkte)

Definieren Sie die Funktion `void TM1637_write_byte(byte wr_data)`, mit der wie oben beschrieben ein Byte `wr_data` an die LED-Segmentanzeige übertragen wird.

Thema: Umgang mit Funktionen und Header-Dateien

3.1.3 LED-Demo (1 Punkt)

Schreiben Sie ein Programm zur Demonstration der LED-Segmentanzeige-Funktionen: Lassen Sie beispielsweise mit zeitlichem Abstand bestimmte Zahlen anzeigen. Nutzen Sie dazu `void TM1637_display_number(float number)` aus `ledanzeige/TM1637.h`.

Hinweis: Rufen Sie **vor** der Arbeit mit der LED-Segmentanzeige **einmal** die Vorgabefunktion `TM1637_setup()` auf. Damit wird die Kommunikation mit der LED-Segmentanzeige initialisiert.

Hinweis: Fügen Sie die Option `-lwiringPi` („minus klein-el“) zu Ihren gcc-Optionen hinzu, damit die Bibliothek beim Linken berücksichtigt wird.

Thema: Compilieren eines Programms, Einbinden von eigenen und Standard-Headern sowie Bibliotheken

3.2 GDB (2 Punkte)

Arbeiten Sie sich selbstständig in die Nutzung des Debuggers GDB (aus dem GCC-Paket) ein.

Demonstrieren Sie in der Abgabe, wie Sie die in den folgenden Aufgaben implementierten Funktionalitäten mit dem GDB von der Konsole aus debuggen können. Dabei müssen Sie mindestens die bereits vom Debuggen unter Java bekannten Funktionalitäten zeigen: Breakpoints setzen und löschen, sich Variableninhalte anzeigen lassen sowie in eine Funktion springen bzw. zum nächsten Breakpoint fortfahren.

Thema: Einarbeitung in den GDB

3.3 Studi-Verwaltung: Verkettete Listen

3.3.1 Strukturen und Datentypen (1 Punkt)

Deklarieren Sie zunächst geeignete Datentypen (`studiverwaltung/studiverwaltung.h`):

a) Für einen Studierenden (Struktur `student`) müssen der Name des Studierenden (Komponente `name`, zu realisieren als `char[]`), die gesammelten Credits (Komponente `cps`) und der eingeschriebene Studiengang (Komponente `enrolled`) erfasst werden.

   Definieren und nutzen Sie die (Präprozessor-) Konstante `NAME_LENGTH` für die maximale Länge der Namen `name`. Die Studiengänge sollen über den Aufzählungstyp `degree_program` abgebildet werden (siehe nächste Teilaufgabe). Was ist ein geeigneter Datentyp für die Credits?

b) Studiengänge: Definieren Sie den Aufzählungstyp `degree_program` mit den Elementen `IFM`, `ELM` und `PFLEGE`.

c) Zur dynamischen Verwaltung sollen doppelt verkettete Listen verwendet werden, in die die Studierenden eingehängt werden. Definieren Sie sich dazu eine Struktur `node`, die einen Pointer auf einen konkreten Studierenden (Typ `student*`, Name `stdnt`) sowie je einen Pointer auf den nächsten und den vorigen Knoten in der Liste (Typ `node*`, Namen `next` und `prev`, wie in der Skizze dargestellt) enthält.

Thema: Definition von Strukturen und Datentypen und Aufzählungen

3.3.2 Listen durchlaufen (2 Punkte)

Schreiben Sie eine Funktion `node* get_ifm_students(node*)`, die eine Hauptliste aller Studierenden als Parameter erhält und eine neue Liste anlegt und zurückliefert, die nur die Informatiker (Studiengang „IFM“) aus der Parameterliste enthält. Ändern Sie dabei nicht die Reihenfolge der Studierenden. Die Studierenden sollen dabei **nicht kopiert**, sondern lediglich in einer weiteren Liste neu verlinkt werden.

Implementieren Sie eine Funktion `void show_all()`, mit der eine solche Liste übersichtlich auf der Konsole ausgegeben werden kann. Die auszugebende Liste (bzw. der Pointer darauf) soll als Funktions-Parameter übergeben werden. Wie muss die Signatur dieser Funktion aussehen?

Thema: Umgang mit Pointern und Iteration durch verkettete Listen

3.3.3 Dynamische Speicherverwaltung (6 Punkte)

Implementieren Sie nun Funktionen zum Aufbauen und Löschen der Studi-Listen.

1) Implementieren Sie eine Funktion `node* append_student(node*, student*)`, die einen neuen Studierenden ans Ende der per Pointer übergebenen Liste einfügt.[1] Wenn der Knoten-Pointer `NULL` ist, soll eine neue Liste angelegt werden. Wenn der Studi-Pointer `NULL` ist, darf die Liste nicht verändert werden.

   Die Funktion liefert einen Pointer auf den (ggf. neuen) Anfang der Liste zurück, falls der Studierende korrekt eingefügt werden konnte, sonst `NULL`.

2) Schreiben Sie eine Funktion `node* delete_node(node*, sp_purge)`, die den Speicher für den übergebenen Knoten freigibt. Wenn als zweiter Parameter `NODE_AND_STUDENT` übergeben wird, dann soll der im Knoten verankerte `stdnt` ebenfalls freigegeben werden (andernfalls bleibt `stdnt` bestehen).

   Falls der zu löschende Knoten in einer Liste verlinkt ist (zu erkennen an den Werten der `prev`- und `next`-Pointer), muss die Funktion die restlichen Knoten der Liste korrekt verketten!

   Die Funktion liefert einen Pointer auf den (ggf. neuen) Anfang der Liste zurück.

   Hinweis: Den Datentyp `sp_purge` finden Sie in der Vorgabe `studiverwaltung/spfree.h`.

3) Schreiben Sie eine Funktion `int delete_list(node*)`, die mit Hilfe von `delete_node()` den **gesamten** von einer Liste belegten Heap-Speicher (inklusive der verwiesenen Studis) wieder freigibt. Rückgabewert ist die Anzahl der freigegebenen Knoten.

4) Schreiben Sie eine weitere Funktion `int delete_list_partial(node*)`, die mit Hilfe von `delete_node()` nur die Knoten der Liste wieder freigibt, d.h. die verwiesenen Studis **nicht** freigibt. Rückgabewert ist die Anzahl der freigegebenen Knoten.

   Diese Funktion braucht man, um von der Hauptliste abgeleitete Hilfslisten („alle Informatiker“) freizugeben.

5) Schreiben Sie ein Programm zur Demonstration der Studi-Verwaltung.

Hinweis: Gehen Sie davon aus, dass der Speicher für die übergebenen Knoten- und Studierenden-Pointer mit `malloc()` alloziert wurde. Arbeiten Sie in den Einfügefunktionen direkt mit den Pointern weiter!

Hinweis: Der jeweils übergebene Pointer `node*` muss nicht auf den Anfang der Liste zeigen. Sie müssen das Startelement selbst suchen (Ausnahme: `delete_node()`, hier zeigt `node*` direkt auf den freizugebenden Knoten). Schreiben Sie sich dazu die Hilfsfunktion `node* get_list_origin(node*)`. Nehmen Sie diese Funktion ausnahmsweise auch mit in die Schnittstelle auf, da wir dafür Testfälle bereit stellen.

Thema: Dynamische Allokation und Freigabe von Speicher, Iteration durch verkettete Listen

3.4 Prüfungsanmeldungen

3.4.1 Funktionen für die Manipulation der Anmeldeliste (3 Punkte)

Implementieren Sie nun die für das Management der Anmeldeliste nötigen Dinge (`pruefungen/pruefungen.[h,c]`):

- Ein Datenfeld `exams` zur Repräsentation der Anmeldeliste (10 Termine).[2] Speichern Sie darin Pointer auf

[1] Normalerweise würde man aus Effizienzgründen den neuen Eintrag vorn in die Liste einhängen. Sie sollen aber hier auch das Traversieren einer verketteten Liste üben.

[2] Wir haben die Übergabe von Arrays noch nicht behandelt: Definieren Sie die Anmeldeliste *ausnahmsweise*(!) als **globale** Variable!

Studierende.

- Die Funktion `int register_student(student *s, int nr)` zum Anmelden des Studierenden `s` für den Termin `nr`. Wenn der Termin bereits belegt ist, wird der Studierende automatisch für den nächsten freien Termin vorgemerkt. Rückgabe ist der Termin (d.h. der Platz/Index in der Anmeldeliste). Falls es keinen freien Termin mehr gab und/oder der Studierende bereits angemeldet war, ist der Rückgabewert -1. Die Anmeldeliste darf in diesem Fall nicht verändert werden. Achten Sie auch auf Null-Pointer (Rückgabe -1).

  Hinweis: Der Parameter `int nr` entspricht dem Index im internen Array `exams` (Werte 0 bis 9)!

- Mit der Funktion `int remove_student(student *s)` kann ein Studierender `s` von der Prüfung zurücktreten, d.h. er wird aus der Liste entfernt und die Rückgabe entspricht seinem bisherigen Termin (Listenplatz). Wenn der Studierende nicht angemeldet war oder nicht existiert (Null-Pointer), ist der Rückgabewert -1.

  Hinweis: Der Studierende selbst darf dabei aber nicht gelöscht werden!

- Die Funktion `float calculate_average(void)` berechnet den Mittelwert der Credits-Points der angemeldeten Studierenden. Das Ergebnis soll in Prozent (Wert zw. 0.0 und 100.0) zurückgeliefert werden, wobei 320 Credits-Points 100 Prozent entsprechen. Definieren Sie die Konstante `MAXIMUM_POINTS` für die 320 cps.

Thema: Umgang mit Arrays, Strukturen und Pointern

3.4.2 Funktionen für die Anzeige der Anmeldeliste (2 Punkte)

Implementieren Sie nun die für die Anzeige des Anmeldestatus nötigen Funktionen (`pruefungen/display.[h,c]`):

- Die Funktion `void display_average(void)` soll den Mittelwert der Credits-Points der angemeldeten Studierenden auf der LED-Anzeige ausgeben.
- Die Funktion `void display_absolute(void)` soll nacheinander für jeden Termin die Nummer des Termins und die Credits-Points des angemeldeten Studierenden (oder den Wert -1, falls Platz frei) ausgeben.

Schreiben Sie ein Programm zur Demonstration der Prüfungsanmeldung.

Thema: C-Funktionen, Iteration durch Arrays, Umgang mit Strukturen, Anzeigen mit der LED-Segmentanzeige