Studiengänge IB, FFI, CSI
WS 2023/24

Technische Hochschule Ingolstadt
Fakultät für Informatik

Praktikum zu „Grundlagen der Programmierung“

Blatt 9

Lernziele: Dynamisch allozierte Arrays, Optionen auf der Kommandozeile
Erforderliche Kenntnisse: Inhalte der Vorlesung zum Thema *Dynamisch allozierte Speicherobjekte*
Voraussetzungen:

- 1. Vollständige Bearbeitung des letzten Blattes 08 (`Worm070`).

Übersicht

Dieses Aufgabenblatt behandelt zwei voneinander unabhängige Themenkomplexe.
Im ersten Teil ersetzen wir das zweidimensionale Array `cells` fester Größe aus der Version Worm070

```
enum BoardCodes cells[MIN_NUMBER_OF_ROWS][MIN_NUMBER_OF_COLS];
```

durch eine Speicherstruktur, die sich in der Benutzung (Zugriff via `cells[y][x]`) wie das zweidimensionale Array verhält, deren Größe aber im Unterschied dazu erst während der Laufzeit festgelegt wird. Dabei werden wir die Speicherstruktur genau so groß wählen, dass das Spielbrett gerade die Größe des Anzeigefensters bekommt (unter Abzug der Message Area am unteren Rand).
Die speziell von uns verwendete Speicherstruktur erlaubt die Modellierung dynamisch allozierter Arrays mit beliebiger Anzahl Dimensionen und Größe. Diese Ausprägung dynamischer Arrays wird unter anderem in der computergestützten numerischen Mathematik verwendet, etwa für Implementierungen von Datenstrukturen und Algorithmen der linearen Algebra (Matrizen, lineare Gleichungssysteme).
Der Wechsel von einer Datenstruktur mit statisch festgelegten Grenzen hin zu einer dynamisch allozierten Datenstruktur mit Grenzen, die erst zur Laufzeit feststehen, zieht einen geringfügig höheren Aufwand bei der Programmierung und etwas mehr Speicherverbrauch nach sich. Wir müssen den Speicherplatz für die Datenstruktur während der Laufzeit explizit anfordern (`malloc`) und bei einer sauberen Programmierung auch wieder explizit freigeben (`free`). Dafür gewinnen wir aber erheblich mehr Flexibilität, die wir zum Beispiel ab Version `Worm090` dazu benutzen werden, dynamisch während der Laufzeit Belegungen des Spielbretts mit Hindernissen und Futterbrocken aus Dateien (`*.level`) nachzuladen, wobei sich das Spielbrett automatisch der Größe des Anzeigefensters anpasst[1].
Ab Version `Worm100` nutzen wir die Technik der dynamischen Speicherallokation, um eine beliebige Anzahl von automatischen Systemwürmern zu implementieren, wobei die gewünschte Anzahl als Option auf der Kommandozeile spezifiziert werden kann.
Der zweite Teil des Aufgabenblatts behandelt die Übergabe von Argumenten an das Programm, die dem Programm beim Aufruf auf der Kommandozeile mitgegeben werden. Wir werden die Funktion

```
int getopt(int argc, char * const argv[], const char *optstring)
```

der C-Standardbibliothek `unistd.h` benutzen, um Argumente von der Kommandozeile einzu-

[1] Allerdings kann die Größe des Anzeigefensters nicht während des Spiels verändert werden. Diese Erweiterung ist zwar mit der Curses-Bibliothek problemlos möglich, verbleibt aber als Fleißaufgabe.

lesen und dadurch das Verhalten des Programms, abhängig von den übergebenen Argumenten (Optionen), unterschiedlich zu konfigurieren.

Aufgabe 1 (Kopieren, Auspacken und Umbenennen des Templates)

Im Moodle-Kursraum finden Sie im Verzeichnis Praktikum die Datei `Worm080Template.zip`. Kopieren Sie diese Datei in Ihr Benutzerverzeichnis `~/GdP1/Praktikum/Code` und öffnen Sie sodann eine Shell. In der Shell wechseln Sie mit dem nachfolgenden Befehl in das Verzeichnis `~/GdP1/Praktikum/Code`:
Vergewissern Sie sich mittels des Befehls ls, dass die Zip-Datei im Verzeichnis vorhanden ist.
`$ ls Worm080Template.zip`
Entpacken Sie die Zip-Datei durch den nachfolgenden Befehl. Dadurch wird das Verzeichnis `Worm080Templat`
mit einigen Dateien darin angelegt.
`$ unzip Worm080Template.zip`
Listen Sie den Inhalt des neuen Unterverzeichnisses.
`$ ls -la Worm080Template`
Benennen Sie das Verzeichnis `Worm080Template` um in `Worm080`. `$ mv Worm080Template Worm080`
Listen Sie den Inhalt des umbenannten Unterverzeichnisses.
`$ ls -la Worm080`
Listen Sie den Inhalt des aktuellen Verzeichnisses.
`$ ls -la`
Löschen Sie das Zip-Archiv.
`$ rm -f Worm080Template.zip`
Führen Sie den nachfolgenden Kopierbefehl[2] aus. Dadurch werden die fünf Dateien

- `board_model.c`
- `messages.c`
- `prep.c`
- `worm.c`
- `worm_model.c`

aus dem Verzeichnis Worm070 in das neue Verzeichnis `Worm080` übernommen (kopiert).
Diese Dateien bilden die Ausgangssituation für die Neuerungen in der Version `Worm080`. `$ cp Worm070/*.c Worm080`

Aufgabe 2 (Test der Ausgangssituation)

Öffnen Sie eine Shell und wechseln Sie in dieser Shell ins Verzeichnis
`~/GdP1/Praktikum/Code/Worm080` Eine Auflistung der Dateien in diesem Verzeichnis sollte

[2]Der Befehl zeigt den Einsatz von Mustern auf der Kommandozeile der Bash-Shell

folgenden Inhalt anzeigen:

```
$ ls
Makefile       board_model.h  options.c  prep.h     worm.h
Readme.fabr    messages.c     options.h  usage.txt  worm_model.c
board_model.c  messages.h     prep.c     worm.c     worm_model.h
```

Bis auf die Dateien mit Dateiendung `.c` (blau dargestellt), die wir zum Schluss der letzten Aufgabe aus dem Verzeichnis der vorherigen Version `Worm070` kopiert haben, sind alle Dateien durch das Auspacken des Templates `Worm080Template.zip` entstanden. Neu sind die beiden Dateien `options.h` und `options.c` (rot dargestellt), mit denen wir uns im zweiten Teil dieses Aufgabenblatts beschäftigen werden. Insbesondere sind alle Header-Dateien im Template geliefert worden. Der Grund hierfür liegt darin, dass damit Ihre mittlerweile vielleicht sehr verschiedenen Lösungsansätze etwas konsolidiert werden. Durch das Kopieren aller `*.c`-Dateien aus dem Verzeichnis `Worm070` wurde jedoch Ihre derzeitige Lösung aus `Worm070` ohne Änderung übernommen. Daher ist Ihre erste Aufgabe nun, Ihre Implementierung (`*.c`) und die aus dem Template stammenden Header-Dateien zu synchronisieren.
Bitte nutzen Sie ein Werkzeug zur Anzeige von Quellcode-Differenzen (`diff` oder grafisch `kdiff3`), um die Unterschiede in Ihren Header-Dateien aus der Version `Worm070` und den vorgegebenen Header-Dateien aus dem Template der Version `Worm080` festzustellen. Falls Sie außer Kommentaren noch andere Unterschiede feststellen, etwa weil Sie Sonderaufgaben Ihres Betreuers implementiert haben oder eigene Modifikationen vorgenommen haben, so können Sie diese Modifikationen nach Belieben in die vorgegebenen Header-Dateien aus dem Template übernehmen.
Am Ende dieser Synchronisierung sollten Sie prüfen, ob Ihr Projekt im Verzeichnis `Worm080` kompilierbar ist. Führen Sie dazu folgenden Befehl im Verzeichnis `Worm080` aus:

```
$ make clean; make
```

Das Programm sollte sich ohne Fehler kompilieren lassen, und auch die Ausführung der Binärdatei `bin/worm` sollte danach, unter Berücksichtigung Ihre persönlichen Modifikationen, fehlerfrei möglich sein. Zum Abschluss dieser Teilaufgabe speichern Sie die Quellen Ihrer initialen lauffähigen Version im Repository. Zum Beispiel so:

```
$ git status; git add .; git commit -m "Worm080 initial"; git push
```

Vorbetrachtung (Vergleich statischer Arrays mit dynamisch allozierten Arrays)

Bevor wir mit der Implementierung des dynamisch allozierten Arrays cells für das Spielbrett beginnen, führen wir uns zunächst einmal den Unterschied zwischen statisch definierten Arrays und der Datenstruktur eines dynamisch allozierten Arrays vor Augen.

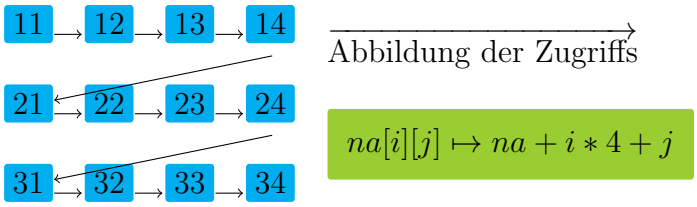
Zweidimensionales Array, herkömmliche statische Definition: Beginnen wir mit der Betrachtung eines normalen zweidimensionalen Arrays na (na für normales Array), welches 3 Zeilen und 4 Spalten hat und Elemente vom Datentyp **int** speichert. Ein derartiges Array sei wie folgt definiert und gleichzeitig explizit initialisiert:

```
int na[3][4] = {
    { 11, 12, 13, 14 },
```

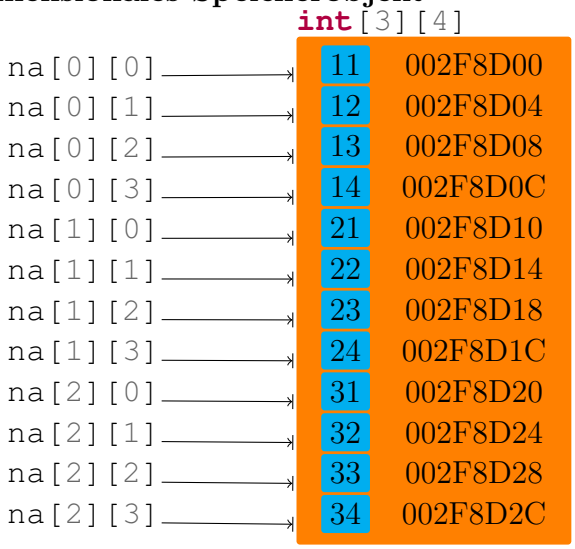
```
    { 21, 22, 23, 24 },
    { 31, 32, 33, 34 },
};
```

Um die Darstellung zu verdeutlichen, sind die gewählten Werte gerade `10 * Zeile + Spalte`, also der Wert in Zeile 2 und Spalte 3 ist 23.
Im Arbeitsspeicher wird dieses Array in einer linearen Anordnung von `3 * 4` Speicherobjekten des Typs **`int`** gespeichert, wobei jedes Speicherobjekt die Anzahl **`sizeof(int)`** Byte benötigt. Auf 32 Bit Maschinen benötigt ein Speicherobjekt vom Typ **`int`** im Allgemeinen 4 Byte. Die 12 Speicherobjekte vom Typ **`int`** für das zu speichernde Array werden vom Compiler laut C-Standard in einem zusammenhängenden Speicherbereich von `12 * 4` Byte angeordnet. Im Beispiel soll das Array an der Speicherposition `002F8D00` beginnen:

Darstellung:
zweidimensionale Struktur



Realisierung:
eindimensionales Speicherobjekt



Beim Zugriff auf das Array an Indexposition `i` für die Zeile und Indexposition `j` für die Spalte wird der Name des Arrays jeweils durch die Basisadresse des Arrays ersetzt. Die Adressierung in Arrays beginnt bekanntlich immer mit dem Index 0. Also im Beispiel $0 \leq i < 4$ und $0 \leq j < 3$. Dann wird nach dem Zugriffsalgorithmus für zweidimensionale statische Arrays das Offset zur Basisadresse als $i * 4 + j$ berechnet.
Wird, wie im Beispiel, auf das Element mit Zeilenindex `i = 1` und Spaltenindex `j = 2` zugegriffen, so errechnet sich die Speicheradresse für den Zugriff wie folgt (Berechnung im Hexadezimalsystem):

```
na[1][2] = na + (1 * 4 + 2) * sizeof(int)
         = 002F8D00 + 6 * 4
         = 002F8D00 + 18
         = 002F8D18
```

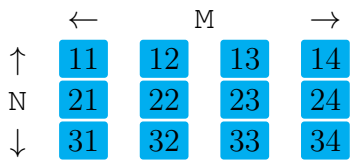
Besonders anzumerken ist, dass für die Speicherung der Basisadresse des Arrays (hier $002F8D00$) kein eigenes Speicherobjekt, also keine Pointer-Variable, angelegt wird! Der Compiler kennt die Basisadresse des Arrays na und ersetzt jedes Auftreten des Array-Bezeichners na durch diese Adresse. Aus diesem Grund kann dem Bezeichner eines statisch definierten Arrays während der Laufzeit keine neue Basisadresse zugewiesen werden.

Zweidimensionales Array, dynamische Allokation (array of pointers to arrays): Die herkömmliche Methode zur Speicherung von statisch definierten Arrays setzt voraus, dass

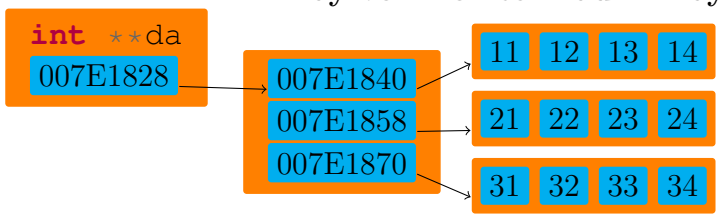
1. alle Elemente des zweidimensionalen Arrays lückenlos im Speicher abgelegt sind.
2. beim Vorgang der Übersetzung bereits die Anzahl der Spalten bekannt ist, da diese in die Berechnung des Offsets $i * 4 + j$ explizit eingeht (im Beispiel 4 Spalten).

Beide genannten Bedingungen stellen eine Einschränkung dar und sind der Grund dafür, dass Arrays mit statisch definierten Größen unbrauchbar sind, wenn die konkrete Anzahl der Zeilen und Spalten erst zur Laufzeit des Programms bekannt sind.
Nun betrachten wir eine alternative Speicherstruktur, für ein zweidimensionales Array mit `N` Zeilen und `M` Spalten, die ohne die beiden Bedingungen realisiert werden kann. Diese Speicherstruktur wird im Allgemeinen als *Array of Pointers to Arrays (im Folgenden APA)* bezeichnet.

Darstellung:
zweidimensionale Struktur



Realisierung:
Array von Pointern auf Arrays



auf Stack — Dynamisch auf Heap alloziert.

Jede Zeile des zweidimensionalen Arrays wird einzeln in einem eindimensionalen Array der Größe `M` gespeichert, das die Spalten der jeweiligen Zeile enthält. Diese eindimensionalen Spalten-Arrays werden dynamisch während der Laufzeit alloziert. Die `N` Anfangsadressen der Spalten-Arrays speichert man in einem eindimensionalen Array der Größe `N`, welches ebenfalls erst zur Laufzeit alloziert wird. Dieses dynamisch allozierte Array von Zeigern hat seinerseits eine Basisadresse, die man sich in einer zusätzlichen Zeigervariablen merkt. Diese letzte einzelne Zeigervariable ist die einzige, die auf dem Stack angelegt wird, das Array von Zeigern und die jeweiligen Spalten-Arrays werden wegen der dynamischen Allokation auf dem Heap abgelegt.
Trotz der erheblich unterschiedlichen Speicherstruktur wird der Zugriff auf ein APA auch als
`da[i][j]`
geschrieben, syntaktisch also genau so wie der Zugriff auf ein herkömmliches Array mit statischen Größen. Der erzeugte Code unterscheidet sich aber deutlich vom üblichen Zugriffscode.
`da[i][j]` → `(da[i])[j]` → `*(*(da + i) + j)`
Der Preis für diese erhöhte Flexibilität, bei gleicher Schreibweise für den Zugriff, ist ein zusätzlicher Speicherplatzbedarf von (`N + 1`) Zeigern, sowie der zusätzliche Code für die Allokation des Speicherplatzes und Freigabe desselben, wenn die Speicherstruktur nicht mehr benötigt wird.
Dieser Preis wird aber im Allgemeinen gerne in Kauf genommen.

Ein C-Programm, welches die hier im Beispiel besprochene Speicherstruktur implementiert, liegt zusammen mit den anderen Dateien dieses Angabenblatts zum Download unter dem Namen `DynamicArrays.zip` bereit.

Studieren Sie den Code dieses Beispiels. Im Rahmen der Teilaufgabe 3 müssen Sie einen ähnlichen

Code im Modul `board_model.c` einfügen. Die Ausführung des Programms erzeugt eine Ausgabe wie folgt, wobei die angezeigten Speicheradressen wahrscheinlich unterschiedlich sein werden. Insbesondere unterscheidet sich die Ausgabe für unterschiedliche Plattformen (Linux/Windows) und Prozessortypen (32/64-Bit) (siehe `mallocArray-2dim.out`, erzeugt in einer Shell):

```
$ bin/mallocArray-2dim
Base pointer da at 0028FF04 points to 007E1828
da[0] at 007E1828
da[0][0] = 11 at 007E1840
da[0][1] = 12 at 007E1844
da[0][2] = 13 at 007E1848
da[0][3] = 14 at 007E184C

da[1] at 007E182C
da[1][0] = 21 at 007E1858
da[1][1] = 22 at 007E185C
da[1][2] = 23 at 007E1860
da[1][3] = 24 at 007E1864

da[2] at 007E1830
da[2][0] = 31 at 007E1870
da[2][1] = 32 at 007E1874
da[2][2] = 33 at 007E1878
da[2][3] = 34 at 007E187C
```

Aufgabe 3 (Dynamisches Array `enum BoardCodes **cell`)

Wechseln Sie ins Verzeichnis des Projekts `Worm080`.
`$ cd ~/GdP1/Praktikum/Code/Worm080`
Öffnen Sie sodann die Header-Datei `board_model.h` in Ihrem Editor. Im Wesentlichen werden wir zwei kleine Änderungen in der Datei `board_model.h` vornehmen:

1. Der Typ der Komponente `cells` der Struktur `struct board` wird geändert. Statt des bisherigen zweidimensionalen Arrays fester Größe
   `enum BoardCodes cells[MIN_NUMBER_OF_ROWS][MIN_NUMBER_OF_COLS];`
   verwenden wir nun nur noch einen Pointer auf eine APA-Struktur
   `enum BoardCodes** cells;`
   Die Komponente `cells` ist nun ein Pointer, der die Adresse eines dynamisch allozierten *Array of Pointers to Arrays*, also einer APA-Struktur, speichert.

2. Wir fügen die Deklaration der Funktion `cleanupBoard` hinzu:
   `extern void cleanupBoard(struct board* aboard);` Diese Funktion werden wir in der Teilaufgabe 4 implementieren. Sie gibt den dynamisch allozierten Speicherplatz, dessen Basisadresse wir in der Komponente `cells` speichern, wieder frei.

Nachdem Sie diese beiden kleinen Änderungen vorgenommen haben, speichern Sie bitte die Header-Datei. Öffnen Sie nun bitte die Datei `board_model.c` in Ihrem Editor und betrachten Sie die

bisherige Implementierung der Funktion `initializeBoard`. Der alte Code sieht in etwa wie folgt aus:

```
enum ResCodes initializeBoard(struct board* aboard) {
    if ( COLS < MIN_NUMBER_OF_COLS  ||
    LINES < MIN_NUMBER_OF_ROWS  + ROWS_RESERVED ) {
        ... hier Code für Fehlermeldung
        return RES_FAILED;
    }
    // Maximal index of a row
    aboard->last_row = MIN_NUMBER_OF_ROWS  - 1;
    // Maximal index of a column
    aboard->last_col = MIN_NUMBER_OF_COLS - 1;
    return RES_OK;
}
```

Bisher haben wir also lediglich aus der festen Anzahl von Zeilen `MIN_NUMBER_OF_ROWS` und Spalten `MIN_NUMBER_OF_COLS` die Werte für den größten erlaubten Zeilenindex und Spaltenindex berechnet und in den Komponenten `last_row` und `last_col` gespeichert. In der neuen Version soll die Größe des Spielbretts sich an der Größe des Anzeigefensters orientieren. Die Anzahl der Zeilen und Spalten des Fensters erhält man bekanntlich über die Curses-Makros `LINES` und `COLS`. Vervollständigen Sie die nachfolgende Code-Schablone, indem Sie die durch `@nnn` gekennzeichneten Stellen ausprogrammieren. Orientieren Sie sich dabei am Beispiel-Code `mallocArray-2dim.c`. Statt des Typs **`int`** müssen Sie natürlich den Typ **`enum`** `BoardCodes` verwenden.

```
enum ResCodes initializeBoard(struct board* aboard) {
    int y;
    // Maximal index of a row, reserve space for message area
    aboard->last_row = LINES - ROWS_RESERVED - 1;
    // Maximal index of a column
    aboard->last_col = COLS - 1;

    // Check dimensions of the board
    if ( aboard->last_col < MIN_NUMBER_OF_COLS - 1 ||
    aboard->last_row < MIN_NUMBER_OF_ROWS - 1) {
        char buf[100];
        sprintf(buf,"Das Fenster ist zu klein: wir brauchen %dx%d",
        MIN_NUMBER_OF_COLS , MIN_NUMBER_OF_ROWS + ROWS_RESERVED );
        showDialog(buf,"Bitte eine Taste druecken");
        return RES_FAILED;
    }
    // Allocate memory for 2-dimensional array of cells
    // Alloc array of rows
    aboard->cells = @001  Hier Speicher allozieren
    if (aboard->cells == NULL) {
        showDialog("Abbruch: Zu wenig Speicher","Bitte eine Taste druecken");
        exit(RES_FAILED); // No memory -> direct exit
    }
    for (y = 0; y < @002; y++) {
        // Allocate array of columns for each y
        aboard->cells[y] = @003 Hier Speicher allozieren
        if (aboard->cells[y] == NULL) {
            @004 Fehlermeldung wie oben
        }
    }
    return RES_OK;
}
```

Im obigen Code werden Sie die Funktion `malloc` zum Allozieren von Speicher verwenden. Aus diesem Grund müssen Sie am Anfang der Modul-Datei `board_model.c` zusätzlich eine `#include`-

Anweisung für die Bibliothek `stdlib.h` angeben:
`#include <stdlib.h>`
Durch die eben vorgenommenen Änderungen passt sich die Größe des Spielbretts nun der Größe des Anzeigefensters an. Übersetzen Sie Ihre aktuelle Version des Programms und testen Sie die Binärdatei. Vergrößern Sie vor dem Start des Programms das Anzeigefenster auf eine Größe von mehr als 30 Zeilen und 70 Spalten. Welchen Unterschied zwischen den Versionen `Worm070` und `Worm080` stellen Sie fest?
Die aktuelle Version des Programms hat noch einen Schönheitsfehler und einen logischen Fehler. Der Schönheitsfehler besteht darin, dass immer noch am rechten Rand des Spielbretts eine senkrechte Linie gezeichnet wird. Dies war in der Vorversion `Worm070` notwendig, da die Größe des Spielbretts fest eingestellt war und kleiner als das Anzeigefenster sein konnte. Nun ist die Begrenzung am rechten Rand nicht mehr notwendig. Entfernen Sie die Anweisungen in der Funktion `initializeLevel`, die bisher die senkrechte Linie am rechten Rand zeichnet und testen Sie dann erneut das ausführbare Programm.
Um den verbleibenden logischen Fehler kümmern wir uns in der nächsten Teilaufgabe.

Bemerkung: Obwohl wir das Spielbrett nun in einem dynamisch allozierten Array speichern (einer APA-Struktur), hat sich der Zugriff auf die Elemente des Arrays syntaktisch nicht geändert. Betrachten Sie dazu den Code der Funktion `placeItem` in der Modul-Datei `board_model.c`. Dort steht nach wie vor:
```
aboard->cells[y][x] = board_code;
```

Aufgabe 4 (Die Funktion `cleanupBoard`)

Der verbleibende logische Fehler besteht darin, dass wir bisher zwar dynamisch Speicherplatz für das Spielbrett mittels `malloc` anfordern, diesen aber nie mehr mittels `free` freigeben.
In unserem Fall mag man argumentieren, dass der Speicherplatz für das Spielbrett nur einmal angefordert wird und am Ende des Spiels (= Ende des ausführenden Prozesses) dann doch automatisch freigegeben wird.
Zum einen ist ein derartiger Programmierstil sehr schlampig und fehleranfällig, denn eine Änderung des Codes dahingehend, dass sich die Größe des Spielbretts einer Änderung des Anzeigefensters während des Spiels anpasst (`resize`) ist gar nicht so weit hergeholt. Wird bei einer derartigen Erweiterung des Codes dann vergessen, die bisher fehlende Speicherfreigabe nachträglich zu kodieren, kann daraus schnell eine hässliche Suche nach einem Speicherleck werden.
Zum anderen ist die Argumentation aber auch falsch, denn ob ein vom Prozess allozierter Speicher nach Beendigung des Prozesses wirklich automatisch freigegeben wird, hängt einzig und allein vom verwendeten Betriebssystem ab. Große Betriebssysteme wie Windows oder Unix/Linux nutzen virtuelle Speicherverwaltung und geben den gesamten, von einem Prozess allozierten, Speicherplatz nach Beendigung des Prozesses frei. Kleinere leichtgewichtigere Betriebssysteme, wie sie zum Beispiel im Embedded-Bereich eingesetzt werden, tun dies aber nicht unbedingt.
Man sollte sich daher immer angewöhnen, jeden dynamisch allozierten Speicher auch explizit wieder freizugeben. Diese Aufgabe erledigen wir nun. Fügen Sie in der Modul-Datei `board_model.c` eine neue Funktion `cleanupBoard` hinzu. Eine Deklaration der Funktion haben Sie bereits in Teilaufgabe 3 der Header-Datei `board_model.h` hinzugefügt.

Hinweis: Orientieren Sie sich bei der Implementierung wieder am Beispiel in `mallocArray-2dim.c`. Dort sehen Sie am Ende der Funktion `main`, wie man den Speicher einer APA-Struktur freigibt. In der Funktion `cleanupBoard` ist die Adresse der APA-Struktur in der Komponente `aboard->cells` gespeichert.

```
void cleanupBoard(struct board* aboard) {
    // Code analog zum Beispiel mallocArray-2dim.c
    ...
}
```

Nachdem Sie die Funktion `cleanupBoard` in der Modul-Datei `board_model.c` implementiert haben, müssen Sie sie natürlich auch noch zum richtigen Zeitpunkt aufrufen. In unserem Spiel ist der richtige Ort dafür die Funktion `doLevel` in der Datei `worm.c`. Dort wird die Strukturvariable `theboard` vom Typ **`struct board`** definiert.

Die Funktion `doLevel` führt ziemlich am Anfang die Anweisung
`initializeBoard(&theboard);`
aus, was zur Allokation der APA-Struktur führt. Am Ende der Funktion `doLevel`, direkt vor der **`return`**-Anweisung, ist ein guter Platz für den Aufruf der Funktion `cleanupBoard`. Fügen Sie dort die Anweisung
`cleanupBoard(&theboard);`
ein.

Nach dieser kleinen Änderung in der Datei worm.c sollte Ihr Projekt wieder übersetzbar sein. Überprüfen Sie diese Annahme durch Ausführung des folgenden Befehls: `$ make clean; make` Das Programm sollte sich ohne Fehler kompilieren lassen, und auch die Ausführung der Binärdatei `bin/worm` sollte fehlerfrei möglich sein.

Hinweis: nun wäre wieder ein geeigneter Zeitpunkt, die aktuelle Version im Repository zu speichern.

Vorbetrachtung (Optionen auf der Kommandozeile)

Nun kommen wir zum zweiten Teil des Aufgabenblatts, in dem wir unser Programm mit Optionen für die Kommandozeile ausstatten. In der Version `Worm080` soll unser Programm folgende Optionen zu Verfügung stellen (siehe auch Datei `usage.txt`):

```
-h   : zeigt Benutzung der Optionen in Kurzform an
-s   : schalte Single-Step schon beim Start ein.
-n s: Zeit s in Millisekunden zwischen zwei Schleifendurchläufen der
      Event-Loop
```

Die Optionen können in beliebiger Reihenfolge benutzt werden und sind, wie der Name schon sagt, optional. Man muss sie also nicht verwenden
Falls die Option `-h` beim Aufruf des Programms verwendet wird, wird angezeigt, wie das Programm aufgerufen werden kann. Nach Betätigung einer Taste wird das Programm dann beendet.

Beispiel:
`$ bin/worm -h`
führt zur Ausgabe
```
Aufruf: worm [-h] [-n ms] [-s]
Bitte eine Taste druecken
```

Wird die Option `-s` verwendet, dann startet das Programm sofort im Einzelschrittmodus. Das ist beim Debugging sehr praktisch. Mit Hilfe der Option `-n` kann man die Verzögerung zwischen den einzelnen Schritten konfigurieren. Damit kann man das Spiel auf verschiedene Prozessorgeschwindigkeiten einstellen. Die normale Geschwindigkeit ist über die CPP-Konstante `NAP_TIME` auf 100 Millisekunden konfiguriert.
Der Aufruf
`$ bin/worm -n 50`
macht das Spiel schneller und stellt die Verzögerung auf nur 50 Millisekunden ein.

Die C-Standardbibliothek stellt die Funktion getopt bereit,
`int getopt(int argc, char * const argv[], const char *optstring)`
mit der Optionen auf der Kommandozeile erkannt und ausgewertet werden können. Vor der Benutzung von getopt muss die Header-Datei `unistd.h` eingebunden werden (siehe C-Manual der Funktion `getopt`).
Optionen auf der Kommandozeile sind in der Praxis sehr wichtig, stehen aber nicht unmittelbar im Fokus Ihrer Ausbildung im ersten Semester. Daher wird Ihnen der Code für die Behandlung von Optionen fertig in Form des Moduls `options.c` nebst Header-Datei `options.h` geliefert. Wir beschränken uns darauf zu erklären, wie Sie das Modul nun in Ihr Programm einbinden können.
In der Header-Datei `options.h` definieren wir eine Struktur **`struct game_options`**. Wir werden später in der Funktion `playGame` (Teilaufgabe 5) eine Variable vom Typ **`struct game_options`** verwenden, um die Einstellungen für die Optionen zu speichern.

```
struct game_options {
    int  nap_time;   // Time in milliseconds to sleep at the end of level loop
    bool start_single_step;  // Start game in single step mode
};
```

Die Struktur enthält zwei Komponenten. Die Variable `nap_time` speichert die Verzögerung, die nach jedem Durchlauf der Hauptschleife in `doLevel` benutzt wird, um das Spiel zu verzögern. Die Komponente `start_single_step` ist eine Boole'sche Variable, die speichert ob der Einzelschrittmodus zu Beginn des Programms aktiv sein soll. In den Versionen `Worm090` und `Worm100` werden wir weitere Optionen hinzufügen.

Aufgabe 5 (Die Funktion `playGame`)

Bisher hat die Funktion `main` direkt die Funktion `doLevel` aufgerufen. Nun ziehen wir hier eine Ebene in Form der Funktion `playGame` ein. Die Funktion `playGame` wird von `main` aufgerufen und ruft ihrerseits dann die Funktion `doLevel` auf. In der Version `Worm080` hat die Funktion `playGame` lediglich die Aufgabe, die Optionen der Kommandozeile zu prüfen und gegebenenfalls darauf zu reagieren. Ab der Version `Worm090` werden wir in der Funktion `playGame` zusätzlich eine Schleife zur Kontrolle der verschiedenen Spielstufen (*Game Level*) implementieren. Das erklärt

dann auch endlich schlüssig den Namen der Funktion doLevel.
Die Steuerung der Spielstufen werden Sie in der Version Worm090 implementieren. Zunächst jedoch soll sich die Funktion playGame nur um die Optionen der Kommandozeile kümmern. Aus oben genannten Gründen wird Ihnen der Code hierfür vorgegeben. Fügen Sie nun folgenden Code der Funktion playGame in die Datei worm.c direkt vor der Funktion main ein:

```
enum ResCodes playGame(int argc, char* argv[]) {
    enum ResCodes res_code; // Result code from functions
    struct game_options thegops; // For options passed on the command line

    // Read the command line options
    res_code = readCommandLineOptions(&thegops, argc, argv);
    if ( res_code != RES_OK) {
        return res_code; // Error: leave early
    }

    if (thegops.start_single_step) {
        nodelay(stdscr, FALSE); // make getch to be a blocking call
    }

    // Play the game
    res_code = doLevel(&thegops);
    return res_code;
}
```

Zu Beginn der Funktion playGame wird eine Strukturvariable thegops vom Typ **struct game_options** erzeugt. Danach wird die Funktion readCommandLineOptions aus der Modul-Datei options.c aufgerufen, die sich um die Behandlung der Kommandozeilenoptionen kümmert. Dort erhält die Komponente nap_time zunächst den Wert, der durch die CPP-Konstante NAP_TIME konfiguriert wird, und start_single_step wird mit false belegt. Werden die Optionen -n und -s auf der Kommandozeile nicht verwendet, dann ändern sich diese Werte nicht. Wird aber eine der Optionen verwendet, dann wird der entsprechende Standardwert in der Strukturvariablen thegops verändert.
Falls der Aufruf der Funktion readCommandLineOptions keinen Fehlercode ungleich RES_OK liefert, wird geprüft, ob das Programm im Einzelschrittmodus starten soll. Dies geschieht durch Abfrage der nun belegten Komponente thegops.start_single_step. Bei Bedarf wird das Programm in den Einzelschrittmodus umgeschaltet.
Danach wird die Funktion doLevel aufgerufen, der die Adresse der Strukturvariablen thegops übergeben wird, damit sie Zugriff auf die eingestellten Optionen erhält. Da die Funktion doLevel in der Version Worm080 auf die Variable doLevel nur lesend zugreift, könnte man selbstverständlich die Strukturvariable thegops auch als Wert übergeben. In der letzten Version Worm100 werden wir aber in der Funktion readUserInput auch schreibend auf die Variable thegops zugreifen und übergeben Sie daher schon jetzt per Adresse an die Funktion doLevel, die die Adresse dann an readUserInput weiterreicht.

Aufgabe 6 (Änderungen bedingt durch Modul **options.c** und Funktion **playGame**)

Nun müssen noch ein paar kleine Änderungen vorgenommen werden, damit das Programm wieder fehlerfrei übersetzt werden kann.

Einbinden der Header-Datei `options.h`: Fügen Sie am Anfang der Datei `worm.c` eine `#include`-Anweisung für die Datei `options.h` ein.

Neuer Parameter für die Funktion `doLevel`: Die Funktion `doLevel` bekommt die Adresse der Variable `thegops` übergeben. Ändern Sie die Funktion `doLevel`, wie im Folgenden skizziert, ab:

```
doLevel(struct game_options* somegops) {
    ...
    napms(somegops->nap_time);
    ...
}
```

Neue Parameter für die Funktion `main`: Die Funktion `main` bekommt von der Umgebung die Anzahl und Liste der Kommandozeilenparameter übergeben. Ändern Sie die Funktion `main`, wie im Folgenden skizziert, ab:

```
int main(int argc, char* argv[]) {
    ...
    res_code = playGame(argc, argv);
    ...
}
```

Rufen sie in der Funktion main nicht mehr wie bisher die Funktion doLevel auf, sondern wie skizziert, die Funktion `playGame`.

Änderungen am `Makefile`: Fügen Sie den Namen der Header-Datei `options.h` und den Namen der Objekt-Datei `options.o` an geeigneter Stelle im `Makefile` hinzu, damit die Modul-Datei `option.c` übersetzt und gebunden wird.

Endkontrolle

Ihr Projekt `Worm050` sollte nun wieder in einem übersetzbaren Zustand sein. Testen Sie diese Annahme durch den folgenden Befehl:
`$ make clean; make`
Das Programm sollte sich ohne Fehler kompilieren lassen, und auch die Ausführung der Binärdatei `bin/worm` sollte fehlerfrei möglich sein. Falls das bei Ihnen nicht so ist, ...
Zum Abschluss speichern Sie die Quellen Ihrer finalen lauffähigen Version im Repository.
Zum Beispiel so:

```
$ git status
$ git add .
$ git commit -m \Worm050 final\
```

Schlußbemerkung

Im Rahmen dieses Aufgabenblatts haben wir die Datenstruktur eines dynamisch allozierten Arrays zur Speicherung des Spielbretts eingeführt, ein sogenanntes Array of Pointers to Arrays (kurz APA). Diese Datenstruktur hat große Vorteile gegenüber den konventionellen Arrays mit fest definierter Größe.
Sie mussten dafür nur sehr wenig Code ändern. Nichts desto trotz gilt es, den Aufbau und die Verwendung dieser Datenstruktur zu verstehen. Nutzen Sie die Chance und studieren Sie die Verwendung von dynamisch allozierten Arrays im Kontext des hier entwickelten Spiels Worm.
Dynamisch allozierte Arrays (APAs) sind ein beliebtes Thema in der Klausur!

Wichtiger Hinweis: Vergessen Sie nicht vor dem Herunterfahren der virtuellen Maschine Ihre in der virtuellen Maschine gemachten Änderungen Ihrem lokalen Repository hinzuzufügen.
Nützliche Kommandos: `git status`, `git add`, `git commit`
Das lokale Repository mit dem Repository bei Bitbucket zu synchronisieren:
Nützliche Kommandos: `git status`, `git push`