Technische Hochschule Ingolstadt
Fakultät für Informatik

Praktikum zu „Grundlagen der Programmierung“

Blatt 10 (Vorführaufgabe)

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

- 1. Vollständige Bearbeitung des letzten Blattes 09 (`Worm080`).

Übersicht

In diesem Aufgabenblatt vertiefen wir das Thema der dynamisch allozierten Arrays, indem wir die Ringstruktur zur Speicherung der Positionen von Wurmelementen umstellen auf dynamisch allozierte eindimensionale Arrays. Dadurch können wir erstens die maximale Größe des Benutzerwurms auf die tatsächliche Größe des Bildschirmfensters anpassen, und zweitens legen wir hiermit den Grundstein für die Erzeugung und Manipulation beliebig vieler automatisch bewegter Systemwürmer.
Die automatischen Systemwürmer werden zwar erst auf Blatt 11 eingeführt, aber die Funktionen zur Erzeugung, Manipulation und Destruktion beliebiger Würmer werden wir im Rahmen des aktuellen Aufgabenblatts vornehmen.
Ein weiteres Thema dieses Blatts ist das Lesen von Spielfeldbeschreibungen aus Dateien, sogenannten Level-Dateien. Ein Level bezeichnet dabei einen bestimmten Schwierigkeitsgrad. Wir verbessern die Ablaufsteuerung für das Spiel, indem wir eine Schleife einführen, die den Spieler von der ersten einfachsten Spielstufe (Level 1) bis hin zur schwierigsten Spielstufe (Level 3) führt. Für jede Spielstufe laden wir eine andere Spielfeldbeschreibung aus einer Datei.
Die Dateien (Level-Dateien), die die Spielfeldbeschreibungen enthalten, können mit jedem Texteditor erstellt oder geändert werden, was Ihrer Kreativität neuen Raum erschließt.
Das vorletzte Kapitel widmet sich dem Thema Zufallszahlen. Hier wird Ihnen gezeigt, wie Sie einen Zufallsgenerator in Ihr Programm einbinden und damit (pseudo-)zufällige Zahlen (ganze Zahlen oder Fließpunktzahlen) in einem bestimmten Bereich erzeugen können.
Mit Hilfe von Zufallszahlen können Sie Ihr Spiel wesentlich interessanter und individueller gestalten. Ideen für optionale Erweiterungen werden im letzten Kapitel dieses Aufgabenblatts aufgezählt. Die möglichen Erweiterungen werden kurz skizziert und sollen Sie zur Erzeugung individueller Varianten Ihres Programms animieren. Gerade die Programmierung von Varianten dient der Vertiefung des im Spiel umgesetzten Lernstoffs und stellt eine optimale Vorbereitung auf die Abschlussprüfung (Klausur) dar.

Aufgabe 1 (Kopieren, Auspacken und Umbenennen des Templates)

Im Moodle-Kursraum finden Sie im Verzeichnis Praktikum die Datei `Worm090Template.zip`. Kopieren Sie diese Datei in das Verzeichnis `~/GdP1/Praktikum/Code`. In einer Shell wechseln Sie mit dem nachfolgenden Befehl in das Verzeichnis `~/GdP1/Praktikum/Code`. `$ cd ~/GdP1/Praktikum/Code`
Vergewissern Sie sich mittels des Befehls `ls`, dass die Zip-Datei im Verzeichnis vorhanden ist.
`$ ls Worm090Template.zip`
Entpacken Sie die Zip-Datei durch den nachfolgenden Befehl.
Dadurch wird das Verzeichnis `Worm090Template` mit einigen Dateien darin angelegt.
`$ unzip Worm090Template.zip`
Listen Sie den Inhalt des neuen Unterverzeichnisses.
`$ ls -la Worm090Template`
Benennen Sie das Verzeichnis `Worm090Template` um in `Worm090`.
`$ mv Worm090Template Worm090`
Listen Sie den Inhalt des umbenannten Unterverzeichnisses.
`$ ls -la Worm090`
Listen Sie den Inhalt des aktuellen Verzeichnisses.
`$ ls -la`
Löschen Sie das Zip-Archiv
`$ rm -f Worm090Template.zip`
Führen Sie den nachfolgenden Kopierbefehl[1] aus. Dadurch werden alle Modul-Dateien (`*.c`) nebst zugehörigen Header-Dateien und das `Makefile` aus dem Verzeichnis `Worm080` in das neue Verzeichnis `Worm090` übernommen (kopiert).
Damit schaffen Sie die Ausgangssituation für die Neuerungen in der Version `Worm090`.
`$ cp ../Worm080/*.c,h ../Worm080/Makefile .`

Aufgabe 2 (Test der Ausgangssituation)

Öffnen Sie eine Shell und wechseln Sie in dieser Shell ins Verzeichnis `~/GdP1/Praktikum/Code/Worm090`. Eine Auflistung der Dateien in diesem Verzeichnis sollte folgenden Inhalt anzeigen:

```
$ ls
Makefile        messages.c            pirates-doubledoom.level.4   usage.txt
Readme.fabr     messages.h            prep.c                       worm.c
basic.level.1   options.c             prep.h                       worm.h
board_model.c   options.h             initializeLevelFromFile.inc  worm_model.c
board_model.h   pirates-doom.level.3  squaredance.level.2           worm_model.h
```

Die blau dargestellten Dateien sind durch das Auspacken des Templates `Worm090Template.zip` entstanden, die anderen Dateien haben Sie zum Schluss der letzten Aufgabe aus dem Verzeichnis der vorherigen Version `Worm080` kopiert.
Stellen Sie sicher, dass Ihr Projekt im Verzeichnis `Worm090` kompilierbar ist. Führen Sie dazu

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

folgenden Befehl im Verzeichnis `Worm090` aus:
`$ make clean; make`
Das Programm sollte sich ohne Fehler kompilieren lassen, und auch die Ausführung der Binärdatei `bin/worm` sollte danach 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 "Worm090 initial"
```

Aufgabe 3 (Einbau der Funktion `initializeLevelFromFile`)

Durch das Entpacken von `Worm090Template.zip` ist auch die Datei `initializeLevelFromFile.inc` entstanden. Die Datei enthält den Code der Funktion `initializeLevelFromFile`, wobei noch ein paar Lücken enthalten sind. Ihrer Aufgabe besteht darin, die Funktion `initializeLevelFromFile` in Ihr Modul `board_model.c` einzubauen und die fehlenden Lücken, wie üblich durch Platzhalter `@nnn` markiert, zu schließen.
Wechseln Sie ins Verzeichnis des Projekts `Worm090`.
`$ cd  /GdP1/Praktikum/Code/Worm090`
Öffnen Sie sodann die Datei `board_model.c` in Ihrem Editor. Suchen Sie die Definition der Funktion `initializeLevel`, die bisher das Spielfeld befüllt hat. Die Funktion `initializeLevel` wird in der Version `Worm090` durch die neue Funktion `initializeLevelFromFile` ersetzt, welche die Beschreibung des Spielfelds (Hindernisse und Futterbrocken) nicht mehr fest vorgibt, sondern aus einer Datei (Level-Datei) einlesen kann.
Positionieren Sie die Schreibmarke direkt vor der Definition der Funktion `initializeLevel` und fügen Sie dort den Code ein, der in der Datei `initializeLevelFromFile.inc` enthalten ist. Falls Sie den Editor *vim* verwenden, erledigen Sie das einfach mittels Befehl:
`:r initializeLevelFromFile.inc`
Dadurch wird der Inhalt der Datei `initializeLevelFromFile.inc` direkt an der Position der Schreibmarke eingefügt. Das `r` steht für *read from file*.
Im Folgenden wird die Funktion `initializeLevelFromFile` in mehreren Ausschnitten dargestellt und kommentiert, damit Sie deren Funktionsweise verstehen und insbesondere auch die Lücken schließen können. Die Funktion `initializeLevelFromFile` verwendet Routinen der C-Standardbibliothek zum Lesen aus Text-Dateien. Diese Thematik wurde im Rahmen der Vorlesung bisher nicht behandelt. Unter anderem wird die Bibliotheksfunktionen `fgets` verwendet, die Sie bereits aus dem C-Tutorium kennen, wo wir sie zum Lesen vom Eingabekanal `stdin` benutzt hatten, um Pufferüberläufe zu vermeiden (siehe Beispiele in `CodeExamples/ReadStrings` und `CodeExamples/ReadStringsAndNumbers`).

```
char *fgets(char *s, int size, FILE *stream);
```

Nun sehen Sie, wie die Funktion `fgets` im allgemeineren Kontext des Lesens von Zeichenketten aus Dateien eingesetzt werden kann. Dabei sollten Ihnen weite Teile des Codes bekannt vorkommen.
Im Folgenden beziehen sich die Zeilennummern auf die Datei `initializeLevelFromFile.inc`. Nach dem Einkopieren in die Modul-Datei `board_model.c` ändern sich natürlich die Zeilennummern.

```
4  enum ResCodes initializeLevelFromFile(struct board* aboard, const char* filename) {
...
8       FILE* in;       // FILE pointer for reading from file
9
10      // Fill board with empty cells.
11      for (y = 0; y <= aboard->last_row; y++) {
12          for (x = 0; x <= aboard->last_col; x++) {
13              placeItem( @001 );
14          }
15      }
```

In Zeile 8 wird ein Zeiger in auf einen Eingabekanal (Eingabestrom) vom Typ **FILE*** definiert. Bisher haben wir immer mit dem vordefinierten Zeiger `stdin` gearbeitet, der für den Standardeingabekanal eines C-Programms steht. In einem späteren Ausschnitt der Funktion `initializeLevelFromFile` werden wir den Kanal in mit einer Datei verbinden und daraus Zeichenketten mittels `fgets` lesen.
In den Zeilen 11–15 sollen alle Zellen des Spielbretts als frei (`FREE_CELL`) markiert werden. Sie müssen noch fehlenden Code an der durch `@001` bezeichneten Stelle einfügen. Studieren Sie hierfür den Code der alten Funktion `initializeLevel` der Version `Worm080`. Deren Code sollte sich auch noch in der von Ihnen gerade bearbeiteten Datei `board_model.c` befinden.

```
22      // --> We need a buffer of size (aboard->last_col+1 + 2 )
23      // The additional 2 elements are for '\n' and '\0'
24      int bufsize = aboard->last_col + 1 + 2;
25      char* buffer;
26      if ((buffer = malloc(sizeof(char) * bufsize)) == NULL) {
27          sprintf(buf,"Kein Speicher mehr in initializeLevelFromFile\n");
28          showDialog(buf,"Bitte eine Taste druecken");
29          return RES_FAILED;
30      }
```

In der Funktion `initializeLevelFromFile` soll die Belegung des Spielbretts zeilenweise aus einer Datei eingelesen werden. Dabei sind pro Zeile nur so viele Zeichen interessant, wie auch im Anzeigefenster des Spiels dargestellt werden können, also `aboard->last_col + 1` Zeichen.
Aus den Beispielen des C-Tutoriums der Vorlesung wissen Sie bereits, dass die Funktion `fgets` in dem für die Speicherung der Eingabe benutzen Zeichenpuffer zusätzlichen Platz für zwei Zeichen benötigt. Zum einen muss das Zeichen für den die Eingabe abschließenden Zeilenvorschub `'\n'` gespeichert werden und zum anderen das Zeichen `'\0'` für die saubere Terminierung der Zeichenkette.
Aus diesem Grund wird in Zeile 24 die Größe des Puffers auf `aboard->last_col + 3` eingestellt und in Zeile 26 ein Eingabepuffer mit Platz für diese Anzahl von Elementen vom Typ **char** alloziert[2].

```
32      // Open the file
33      if ( (in = fopen(filename,"r")) == NULL) {
34          sprintf(buf,"Kann Datei %s nicht oeffnen",filename);
35          showDialog(buf,"Bitte eine Taste druecken");
36          return RES_FAILED;
37      }
```

[2] An dieser Stelle machen wir uns gedanklich einen Knoten ins Ohr oder, damit es nicht so weh tut, einen Knoten in ein Taschentuch, damit wir später daran denken, diesen allozierten Speicher wieder mittels `free(buffer)` frei zu geben.

In Zeile 33 öffnen wir mittels `fopen` die Datei, deren Namen über den Parameter `filename` der Funktion `initializeLevelFromFile` übergeben wird. Das `"r"` spezifiziert, dass wir aus der Datei lesen möchten.

```
FILE *fopen(const char *path, const char *mode);
int fclose(FILE *fp);
```

Ab dieser Stelle können wir auf die Datei über den Zeiger `in` vom Typ **`FILE`**`*` lesend zugreifen, ganz so wie bisher in den Beispielen über den Zeiger `stdin`.
Ähnlich wie auf `malloc` immer auch ein passendes `free` folgen sollte, muss analog auf ein `fopen` immer ein entsprechendes `fclose` folgen, das den geöffneten Strom, egal ob Eingabe oder Ausgabe, wieder sauber schließt.
Da nun der Eingabekanal zum Lesen geöffnet ist, können wir mittels `fgets` die einzelnen Zeilen der Level-Datei einlesen. Wir initialisieren in Zeile 39 einen Zähler `rownr` für die aktuelle Zeilennummer und betreten eine **`while`**-Schleife, die Zeile für Zeile aus der Eingabedatei liest, aber höchstens solange, bis entweder die maximale Anzahl der für das Spielbrett reservierten Zeilen (`aboard->last_row+1`) erreicht ist, oder keine weiteren Zeilen mehr aus der Datei gelesen werden können. Der Aufruf der Funktion `feof` prüft, ob das Ende des Datenstroms in erreicht ist.

```
39      rownr = 0;
40      // Read all lines from the text file describing the level
41      while (rownr < aboard->last_row+1 && ! feof(in)) {
42          int len;
43          // Read one line from file
44          if (fgets(buffer,bufsize,in) != NULL) {
...
49              len = strlen(buffer);
50              // Note: function strlen does not count the terminating '\0'.
51              if (buffer[len -1] != '\n') {
52                  // Input line was not yet finished. No '\n' in buffer.
53                  // Delete the last char before '\0'from the buffer because
54                  // it exceeds the allowed number of characters.
55                  buffer[len -1] = '\0';
56                  // Delete the rest of the line that is . . .
57                  while (fgetc(in) != '\n'){;}
58              } else {
59                  // Input line in buffer ends with '\n'
60                  // Delete the '\n' from the buffer
61                  buffer[len -1] = '\0';
62              }
```

In Zeile 44 wird eine ganze Zeile aus dem Eingabestrom in den Puffer `buffer` gelesen. Das Zeilenende in der Eingabedatei wird durch das Zeichen `'\n'` (ASCII-Code 10 für *Linefeed*) kodiert. Die Funktion `fgets` garantiert, dass nicht mehr als `bufsize` Zeichen des Puffers belegt werden, wobei zusätzlich auch Platz für das Terminierungszeichen `'\0'` reserviert wird. Falls die aus der Datei eingelesene Zeile weniger Zeichen enthält als im Puffer Platz finden, dann wird das abschließende Zeichen der Zeile `'\n'` in den Puffer übernommen und der Puffer dann sauber mit `'\0'` terminiert, damit der Pufferinhalt als Zeichenkette weiterverarbeitet werden kann. Wenn aber die Eingabezeile der Datei mehr Zeichen enthält, als der Puffer fassen kann, dann werden nur `bufsize - 1` Zeichen aus der Datei gelesen und der letzte freie Platz im Puffer mit `'\0'` belegt. Die Funktion `fgets` schützt also vor Pufferüberlauf und liefert garantiert eine durch `'\0'` terminierte Zeichenkette im Puffer zurück.
Falls `fgets` erfolgreich eine Zeile einlesen konnte (Test Zeile 44 auf `!= NULL`), dann liegt in

`buffer` auf jeden Fall eine sauber terminierte Zeichenkette, deren Länge mittels `strlen` berechnet werden kann. Falls der Puffer groß genug für die Aufnahme der ganzen Zeile war, so muss das letzte Zeichen der Zeichenkette in `buffer` gerade das Code-Zeichen `'\n'` sein. Dies wird in Zeile 51 geprüft.
Der Code in den Zeilen 55 und 57 ist von entscheidender Bedeutung für die korrekte Verarbeitung der gerade gelesenen Eingabezeile. Der Code dieser Zeilen wir nur ausgeführt, wenn das letzte Zeichen vor dem terminierenden `'\0'` kein `'\n'` war. In diesem Fall enthält die aktuelle Eingabezeile der Datei mehr Zeichen, als der Puffer aufnehmen kann. Der Puffer `buffer` fasst maximal `aboard->last_row + 3` Zeichen. Das letzte Zeichen im Puffer ist das terminierende `'\0'`, und davor liegen aktuell `aboard->last_row + 2` Zeichen im Puffer. Das ist ein Zeichen mehr, als wir in unserem zweidimensionalen, dynamisch allozierten Array für das Spielfeld pro Zeile speichern können. Aus diesem Grund wird in Zeile 55 das Zeichen an Position `len - 1` durch ein `'\0'` ersetzt, was die Zeichenkette auf die maximal mögliche Länge einer Zeile des Spielfeldes verkürzt.
Da die aktuelle Zeile mehr Zeichen umfasste, als `fgets` einlesen konnte, befinden sich die überzähligen Zeichen nun im einem Zwischenpuffer der Ein-/Ausgabebibliothek. Dort werden sie bis zum nächsten Aufruf einer Eingabefunktion, etwa `fgets`, gespeichert und dieser dann als Eingabe zur Verfügung gestellt. Diese Zwischenpufferung ist in unserem Fall aber nicht erwünscht, denn beim nächsten Aufruf der Funktion `fgets` in unserer Eingabeschleife wollen wir natürlich nicht den Rest der zu langen Eingabezeile lesen, sondern die nächste Zeile der Level-Datei.
Der Code in Zeile 57 dient dazu, die überflüssigen Zeichen loszuwerden. Mittels Funktion `fgetc`

```
int fgetc(FILE *stream);
```

lesen wir solange einzelne Zeichen aus dem Zwischenpuffer der Ein-/Ausgabebibliothek, bis wir das Ende der aktuellen Eingabezeile der Level-Datei erreicht haben. Alle dabei gelesenen Zeichen sollen ignoriert werden. Teilaufgabe 5 wird plakativ verdeutlichen, was passieren würde wenn wir den Zwischenpuffer nicht, wie in Zeile 57, leerten. Die Zeile 57 ließe sich noch kürzer wie folgt formulieren:
`while (fgetc(in) != '\n');` Wir haben jedoch absichtlich die Syntax `{;}` gewählt, um deutlicher auf die leere Anweisung hinzuweisen.
Der Code in Zeile 61 wird nur ausgeführt, wenn die aktuelle Eingabezeile vollständig, inklusive des abschließenden Zeilenumbruchs `'\n'`, in den Puffer buffer passt. In diesem Fall löschen wir das Zeichen `'\n'` für den Zeilenumbruch, da wir kein Interesse an seiner Speicherung haben. Wir ersetzen daher das überflüssige Zeichen durch `'\0'` und verkürzen so die Zeichenkette im Puffer `buffer`.

```
63          } else {
64              // fgets returned NULL
...
67              if (!feof(in)) {
...
72                  return RES_FAILED;
73              } else {
74                  // We got EOF, skip rest of loop
75                  // Loop condition will fire
76                  continue;
77              }
78          }
```

Die Zeilen 63-78 behandeln den Fall, dass der Aufruf von `fgets` einen *NULL-Pointer* zurückgegeben hat. Das kann zwei Gründe haben. Entweder haben wir das Ende der Datei erreicht, was keine Fehlersituation darstellt oder aber es liegt ein echter Lesefehler vor. Die beiden Fälle unterscheiden

wir mittels Bedingung `! feof(in)` in Zeile 67. Falls ein Fehler vorliegt, beenden wir sofort die Funktion mit einem Fehlercode, im anderen Fall beenden wir nur den aktuellen Schleifendurchlauf und fahren direkt mit der Prüfung der Schleifenbedingung fort (**`continue`**). Dort wird bei Eintreten von `feof(in)` die Schleife zum Einlesen aller Zeilen der Level-Datei beendet.

```
85          for (x = 0; x <= aboard->last_col && x < len; x++) {
86              switch (buffer[x]) {
87                  case SYMBOL_BARRIER:
88                      placeItem(aboard,rownr,x,BC_BARRIER,
89                                SYMBOL_BARRIER,COLP_BARRIER);
90                      break;
91                  case SYMBOL_FOOD_1:
92                      @002
93                  case SYMBOL_FOOD_2:
94                      @002
95                  case SYMBOL_FOOD_3:
96                      @002
97
98                  // We ignore all other symbols!
99              }
100         }
101         // advance to next input line
102         rownr++;
103     } // END while
```

Der Code ab Zeile 79 wird ausgeführt, wenn die aktuelle Eingabezeile erfolgreich eingelesen werden konnte. An der Stelle ist die aktuelle Eingabezeile aus der Level-Datei, eventuell in gekürzter Form, im Puffer `buffer` gespeichert. Ab Zeile 85 verarbeiten wir in einer Schleife alle im Puffer enthaltenen Zeichen. Die Fallunterscheidung (**`switch`**) ab Zeile 86 unterscheidet dabei die einzelnen Eingabezeichen, für die wir uns interessieren. Je nach dem, welches Zeichen wir im Puffer vorfinden, speichern wir den entsprechenden Board-Code in unserer Datenstruktur für das Spielbrett und das Zeichen (`SYMBOL_`$\star$) im Fensterpuffer für die Curses-Ausgabe ab. Wir verwenden dazu, wie üblich, die Funktion `placeItem` (siehe Zeile 88).
Nachdem Sie bisher fast nur vorgefertigten Code der Funktion `initializeLevelFromFile` verstehen mussten, müssen Sie nun aktiv werden und die fehlenden Zeilen, an den durch den Platzhalter `@002` bezeichneten Stellen, ergänzen. Orientieren Sie sich dabei an der Zeile 88 und an Ihrem alten Code in der Funktion `initializeLevel`.

```
104     // Free the line buffer
105     free(buffer);
106
107     // Draw a line in order to separate the message area
...
110     for (x=0; x <= aboard->last_col; x++) {
111         @003
112     }
113
114     fclose(in);
115     return RES_OK;
116 }
```

Nachdem die Schleife zum Einlesen aller Zeilen der Level-Datei beendet ist, müssen wir noch ein bischen aufräumen und die Initialisierung des Fensterpuffers abschließen. In Zeile 105 erinnern wir uns an den Knoten, den wir bei der Kodierung der Zeile 26, wohin auch immer, gemacht haben. Wir geben den für das Einlesen der Zeilen allozierten Puffer wieder frei. Ähnlich erinnern wir uns, dass die in Zeile 33 geöffnete Datei wieder geschlossen werden muss. An der in Zeile 111 durch `@003` markierten Stelle müssen Sie ein paar Zeilen einfügen, die die Trennlinie zur Message-Area

ziehen (siehe alte Funktion `initializeLevel` aus Version `Worm080`).
Hiermit ist die Funktion `initializeLevelFromFile` vollständig besprochen, und Sie haben die Anwendung einiger wichtiger Funktionen der C-Bibliothek zum Lesen von Dateien im Kontext einer Ihnen geläufigen Anwendung kennengelernt. Die Betrachtung im Anwendungskontext ist dabei wesentlich hilfreicher, als die isolierte Darstellung der einzelnen Funktion anhand trivialer Code-Beispiele in der Vorlesung.
Zum Abschluss der Integration der neuen Funktion `initializeLevelFromFile` in die Modul-Datei `board_model.c` müssen Sie noch die Header-Datei der String-Bibliothek `string.h` zu Beginn der Datei `board_model.c` einfügen:

```
#include <string.h>
```

Desweiteren müssen Sie in der Header-Datei `board_model.h` noch eine Deklaration der neuen Funktion `initializeLevelFromFile` wie folgt einfügen:

```
extern enum ResCodes
initializeLevelFromFile(struct board* aboard, const char* filename);
```

Wenn Sie möchten, können Sie nun die nicht mehr benötigte Funktion `initalizeLevel` aus der Modul-Datei `board_model.c` und aus der Header-Datei `board_model.h` löschen.

Aufgabe 4 (Benutzung der neuen Funktion `initializeLevelFrom`

Natürlich möchten wir nun unsere neue Funktion `initializeLevelFromFile` auch im Programm nutzen, um Level-Dateien einzulesen. In dieser Teilaufgabe machen wir einen einfachen Anfang und lesen mittels `initializeLevelFromFile` zunächst immer die spezielle Datei `pirates-doubledoom.level.4` ein.
Die Einschränkung auf diese eine Datei werden wir dann in Teilaufgabe 16 aufheben, nachdem wir in Teilaufgabe 5 noch ein lehrreiches Experiment anhand der Datei `pirates-doubledoom.level.4` durchführen werden.
Öffnen Sie nun bitte die Datei `worm.c` in einem Editor und ersetzen Sie den bisherigen Aufruf

```
res_code = initializeLevel(&theboard);
```

in der Funktion `doLevel` durch

```
res_code = initializeLevelFromFile(&theboard, "pirates-doubledoom.level.4");
```

Ihr Projekt `Worm090` 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. Im Gegensatz zu früher sehen Sie nun aber nicht mehr das alte fest kodierte Spielfeld, sondern das in der Datei `pirates-doubledoom.level.4` spezifizierte Spielfeld.
Probieren Sie Ihr Programm mit verschiedenen Größen des Anzeigefensters aus, unter anderem auch mit einer Fenstergröße von mehr als 100 Spalten und mehr als 30 Zeilen.
Hinweis: nun wäre ein guter Zeitpunkt, einen Zwischenstand Ihres Codes im Repository zu speichern.

Aufgabe 5 (Löschen der überzähligen Zeichen im Zwischenpuffer)

Stellen Sie Ihr Anzeigefenster auf eine Größe von 80 Spalten und 35 Zeilen ein. Führen Sie sodann das Programm `bin/worm` aus. Das Spielfeld der Level-Datei `pirates-doubledoom.level.4` wird dargestellt, allerdings werden die Eingabezeilen der Datei auf die Größe des Anzeigefensters angepasst und nach der Spalte 85 abgeschnitten. Genau so hatten wir das im Code der Funktion `initializeLevelFromFile` festgelegt.
Öffnen Sie die Level-Datei `pirates-doubledoom.level.4` in einem Editor und betrachten Sie die Datei in ihrer vollen Breite von 141 Zeichen. Eventuell müssen Sie dazu den Font kleiner einstellen.
Bei der Besprechung der Zeilennummer 57 der Datei `initializeLevelFromFile.inc`[3]

```
56                  // Delete the rest of the line that is ...
57                  while (fgetc(in) != '\n'){;}
```

wurde argumentiert, dass die überzähligen Zeichen von Zeilen, die zu lange für den Eingabepuffer sind, explizit durch die entartete **while**-Schleife konsumiert werden müssen, damit sie nicht bei anschließenden Aufrufen der Funktion `fgets` stören.
In dieser Teilaufgabe probieren wir einfach aus, was passiert wenn man den Code in Zeile 57 deaktiviert.
Öffnen Sie erneut die Datei `board_model.c` und kommentieren Sie die **while**-Schleife

```
// Delete the rest of the line that is ...
// while (fgetc(in) != '\n'){;}
```

in der Funktion `initializeLevelFromFile` aus. Natürlich finden Sie den Code nicht in der Zeile 57, denn in der Datei `board_model.c` befindet sich der Code der Funktion `initializeLevelFromFile` wahrscheinlich an einer ganz anderen Stelle.
Übersetzen Sie Ihr Programm erneut und testen Sie bin/worm mit den gleichen Einstellungen für die Fenstergröße wie vorher.

- Welchen Unterschied stellen Sie fest?
- Können Sie das geändert Verhalten einem Kommilitonen erklären?

Aktivieren Sie zum Schluss wieder den eben auskommentierten Code.

Aufgabe 6 (Name der Level-Datei als Argument auf Kommandozeile)

In dieser Teilaufgabe erweitern wir die Implementierung dahin gehend, dass man beim Aufruf der ausführbaren Datei den Namen einer beliebigen Level-Datei auf der Kommandozeile angeben kann. Bisher können wir auf der Kommandozeile die Optionen `-h`, `-n` und `-s` angeben (siehe `usage.txt`). Nun soll als zusätzliches Argument der Name einer Level-Datei angegeben werden können, wobei

[3] die Zeilennummern beziehen sich, wie bei der Besprechung, auf `initializeLevelFromFile.inc`

vor dem Dateinamen kein Optionsschalter, also kein `'-'` gefolgt von einem Buchstaben, angegeben werden soll. Natürlich könnte man zum Beispiel den Optionsschalter `-f` nach dem gleichen Prinzip wie die anderen Optionsschalter verwenden, aber Sie sollen hier eine alternative Lösung kennen lernen.
Der Name der Level-Datei soll von der Kommandozeile gelesen werden können. Daher fügen wir dem Verbund **`struct game_options`** einen Zeiger auf eine Zeichenkette hinzu. Der Name des Zeigers soll `start_level_filename` sein. Öffnen Sie die Datei `options.h` und fügen Sie die folgende Komponente hinzu:

```
struct game_options {
        ...
        char* start_level_filename;
}
```

In der Modul-Datei `options.c` müssen wir nun das Einlesen und Speichern des Namens der Level-Datei implementieren. Öffnen Sie die Datei options.c und führen Sie folgende Änderungen durch.
Die Ausgabe des Hilfetextes muss angepasst werden:

```
void usage() {
        char buf[100];
        sprintf(buf,"Aufruf: worm [-h] [-n ms] [-s] [ Dateiname ]");
        showDialog(buf,"Bitte eine Taste druecken");
}
```

Im üblichen Sprachgebrauch werden die auf der Kommandozeile eines Programms aufgeführten Zeichenketten in zwei Gruppen eingeteilt. Zeichenketten, die mit einem `'-'` eingeleitet werden und von einem (oder auch mehreren) Buchstaben gefolgt werden, heißen Optionen (*Options*) oder Optionsschalter (*Switches*). Solchen Optionen darf auch ein Wert folgen. Beispiele für Optionsschalter sind:

```
-d\\
-n 10\\
-f blabla
```

Der Optionsschalter `-d` hat keinen zusätzlichen Wert und fungiert als Boole'scher Schalter. Der Optionsschalter `-n` hat den Integer 10 als Wert und der Optionsschalter `-f` hat die Zeichenkette `blabla` als Wert.
Alle anderen Zeichenketten auf der Kommandozeile werden als Argumente des Programms bezeichnet.
Der Grund für die Trennung liegt darin, dass die Optionsschalter in beliebiger Reihenfolge in der Kommandozeile angegeben werden dürfen, da sie ja am jeweiligen Optionsschalter erkannt werden können.
Die restlichen Argumente der Kommandozeile hingegen sind nicht speziell markiert (kein Optionsschalter) und müssen daher meist über deren Position erkannt werden.

Technisch wird zuerst die ganze Kommandozeile untersucht, wobei alle Optionsschalter nebst den zugehörigen Werten interpretiert werden. Danach werden die normalen Argumente verarbeitet, die dann anhand ihrer relativen Position zueinander identifiziert werden.
Wir verwenden die Funktion `getopt` aus der C-Standardbibliothek.

```
#include <unistd.h>
int getopt(int argc, char * const argv[], const char *optstring);
```

um Optionen und Argumente der Kommandozeile zu lesen. Im Folgenden werden Sie im Anwendungskontext die Verwendung der Funktion `getopt` erlernen. Details erfahren Sie, zumindest unter Unix/Linux, über die man-Page der Funktion mittels folgenden Aufruf:
`man -s 3 getopt`
Das `'-s3'` steht für die Sektion 3 des Online-Manuals, das auf jedem ordentlich installierten Unix/Linux-System zur Verfügung steht. In unserer Modul-Datei `options.c` haben wir die Funktion `readCommandLineOptions` implementiert, die sich um die Verarbeitung der Optionen und Argumente auf der Kommandozeile kümmert.
Am Anfang der Funktion werden die Komponenten des Verbunds **`struct game_options`** initialisiert. Hier müssen wir nun die neue Komponente `start_level_filename` initialisieren.

```
// Initialize;
somegops->nap_time = NAP_TIME;
somegops->start_single_step = 0;
somegops->start_level_filename = NULL;
```

Nun mögen Sie sich vielleicht fragen, welchen Sinn die Initialisierung des Pointers mit dem Wert `NULL` hat. Die Antwort ist ganz einfach: Die Angabe eines Namens einer Level-Datei auf der Kommandozeile ist optional, d.h. es kann vorkommen, dass keine Datei angegeben wird. Dieser Fall ist sogar der Normalfall und wird dadurch explizit gemacht, dass der Pointer `start_level_filename` dann auf keine Zeichenkette zeigt, die als Dateiname interpretiert werden könnte.
Da wir in der Funktion `readCommandLineOptions` den Pointer
`struct game_options``* somegops`
als Funktionsargument bekommen, wissen wir nicht, wie dessen Komponente `start_level_filename` belegt ist. Daher ist es wichtig, den Pointer hier mit `NULL` zu belegen.
In der **`while`**-Schleife wird die Funktion `getopt` aufgerufen, die sich einen Optionsschalter nach dem anderen, wie die Rosinen aus dem Kuchen, aus der Kommandozeile herauspickt. Die Optionsspezifikation `n:s` bedeutet, dass der Optionsschalter `-n` ein Argument hat (`':'`) und der Schalter `-s` kein Argument hat.

```
while((c = getopt(argc, argv, "n:s")) != -1)
switch(c) {
        ...
        case('n'):
        somegops -> nap_time = atoi(optarg);
        continue;
        case('s'):
        somegops -> start_single_step = true;
        continue;
        ...
        Der Name der Level-Datei soll als Argument und nicht als Option auf der Kommandozeile
        // Skip all options processed
        argc -= optind;
        argv += optind;
        // Are there any non-option arguments left?
        // In our case at most one argument is allowed
        if (argc > 1 ) {
                usage();
                return RES_WRONG_OPTION;
        }
        // The argument is supposed to be the filename of a level description
        if (argc == 1) {
                somegops -> start_level_filename = strdup(argv[0]);
```

```
            // Do not forget to free the allocated memory somewhere else!
        }
```

Falls die Kommandozeile genau ein weiteres Argument enthält, speichern wir dieses Argument, indem wir die Zeichenkette, auf die nun `argv[0]` zeigt, mittels `strdup` kopieren. Der Aufruf der Funktion `strdup`

```
#include <string.h>
char *strdup(const char *s);
```

alloziert dynamisch Speicher für die Kopie der Zeichenkette, kopiert diese und liefert den Zeiger auf das im Heap angelegte Speicherobjekt. Wir merken uns wieder, dass wir diesen allozierten Speicher später mittels `free` freigeben müssen!
Damit haben Sie alle notwendigen Änderungen im Modul `options` ausgeführt, die zur Verarbeitung des neuen Kommandozeilenarguments notwendig sind. Mehr Informationen zum Umgang mit Optionen erfahren Sie, wie bereits erwähnt, über die man-Page der Funktion `getopt` mit dem Aufruf:
`man -s 3 getopt`
Das neue optionale Argument der Kommandozeile, der Name der Level-Datei, wird nun schon eingelesen und in der Strukturvariablen **`struct game_options`** `thegops` gespeichert, die in der Funktion `playGame` definiert wird. Im Hinblick auf die spätere Implementierung einer Schleife für mehrere, im Schwierigkeitsgrad steigende, Spielstufen (Level) programmieren wir zum Abschluss dieser Teilaufgabe eine Übergangslösung, die wir dann in der nächsten Teilaufgabe durch die finale Logik für mehrere Spielstufen ersetzen werden. Beide Lösungen haben gemeinsam, dass die Funktion `playGame` beim Aufruf der Funktion `doLevel` dieser den Namen der zu ladenden Level-Datei mitgeben soll.
Aus diesem Grund ändern wir nun zuerst die Signatur1 der Funktion `doLevel` und fügen folgenden Code hinzu:

```
enum ResCodes doLevel(struct game_options* somegops, char* level_filename) {
        ...
        // Initialize the current level
        res_code = initializeLevelFromFile(&theboard, level_filename);
```

Nun müssen wir in der Funktion `playGame` nur noch dafür sorgen, dass eine spezielle Level-Datei eingelesen wird, wenn deren Name auf der Kommandozeile spezifiziert wurde. Als letzte Änderung dieser Teilaufgabe ändern wir deswegen die Logik für den Aufruf der Funktion `dolevel` in der Funktion `playGame`.
Diese Lösung ist ein Provisorium, das dann in der Teilaufgabe 7 um eine Schleife für mehrere Spielstufen erweitert wird. Ändern Sie den Code in der Funktion `playGame` wie folgt ab und ergänzen Sie die mit `@nnn` blau markierten Lücken:

```
        if (thegops.start_single_step) {
                nodelay(stdscr, FALSE);  // make getch to be a blocking call
        }
        // Play the game
        if (thegops.start_level_filename != NULL) {
                // User provided a filename on the command line.
                // Play only this level
                res_code = doLevel(&thegops, @010 );

                // From here on we no longer need thegops.start_level_filename
                // Free the memory allocated by strdup in options.c
```

```
                free( @020 );
        } else {
                // Play standard level
                res_code = doLevel(&thegops, "basic.level.1" );
        }
        return res_code;
}
```

Ihr Projekt `Worm090` 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. Testen Sie insbesondere den Aufruf des Programms mit unterschiedlichen Level-Dateien, also zum Beispiel:

```
$ bin/worm -n 50 squaredance.level.2
$ bin/worm -n 200 -s pirates-doom.level.3
```

Testen Sie das Programm auch mit selbst entworfenen Level-Dateien!

Aufgabe 7 (Schleife für mehrere Spielstufen)

In dieser Teilaufgabe erweitern wir die Steuerung des Spiels so, dass mehrere Schwierigkeitsstufen (Levels) hintereinander gespielt werden müssen, wobei mit der einfachsten Stufe begonnen wird, und nach jeder fehlerfreien Absolvierung einer Spielstufe das Spiel mit der nächst höheren Spielstufe fortgesetzt wird. Für jede Spielstufe soll eine, der Schwierigkeitsstufe entsprechende, Spezifikation des Spielbretts aus einer Level-Datei geladen werden.
Die für diese Erweiterung notwendigen Änderungen fallen im Wesentlichen in der Funktion `playGame` an. Desweiteren sind ein paar kleine Änderungen in der Funktion `doLevel` und im Modul `worm_model` notwendig.
Die Steuerung der neuen Schleife über alle Spielstufen soll in der Funktion `playGame` liegen. Da während der Ausführung eines Levels durch die Funktion `doLevel` unterschiedliche Fahrfehler oder auch der Wunsch nach einem Abbruch durch den Benutzer erfolgen können, muss der Zustand des Spiels (**enum** `gameStates`) von `doLevel` zur aufrufenden Funktion `playGame` kommuniziert werden. Aus diesem Grund verschieben wir zu aller erst die Variable **enum** `GameStates game_state` aus der Funktion `doLevel` in die Funktion `playGame` und geben der Funktion `doLevel` dafür einen Pointer auf diese Variable als neues Argument mit.
Führen Sie folgende Änderungen an den Funktionen `doLevel` und `playGame` durch.

1. Ändern Sie die Signatur der Funktion `doLevel` wie folgt ab:

   ```
   enum ResCodes doLevel(struct game_options* somegops,
   enum GameStates* agame_state, char* level_filename)
   ```

2. Verschieben Sie die Definition der Variablen

   ```
   enum GameStates game_state; // The current game_state
   ```

aus der Funktion `doLevel` in die Funktion `playGame`.

3. Verschieben Sie die Initialisierung der Variablen `game_state` aus der Funktion `doLevel` in die Funktion `playGame`.

4. Ändern Sie alle Aufrufe der Funktion `doLevel` in der Funktion `playGame` entsprechend ab.

5. Ändern Sie entsprechend in `doLevel` alle Zugriffe auf die Variable `game_state` in Zugriffe über den Pointer `agame_state` ab.

Nach dieser Änderung sollte Ihr Projekt wieder kompilierbar sein und fehlerfrei ausgeführt werden können.
Nun führen wir in der Funktion `playGame` eine neue Liste (Array `level_list`) von Namen für Level-Dateien ein und fügen auch eine Schleifenvariable `cur_level` für die Schleife über alle Spielstufen hinzu. Ergänzen Sie Ihre Funktion `playGame` um folgende Variablendefinitionen:

```
// An array of filenames for level descriptions
// The list must be terminated by a NULL pointer!
char* level_list[] = {
        "basic.level.1",
        "squaredance.level.2",
        "pirates-doom.level.3",
        NULL
};
int cur_level;       // The current level
```

Für das Array `level_list`, welches ein statisch alloziertes Array von Pointern auf konstante Zeichenketten ist[4], verabreden wir die Konvention, dass das Ende des Arrays durch einen `NULL`-Pointer signalisiert werden soll. Dadurch spart man sich eine Variable oder Konstante für die letzte belegte Position im Array und kann Schleifen dementsprechend kompakter schreiben.
Nun bauen wir das Provisorium aus Teilaufgabe 6 aus und integrieren im **`else`**-Zweig stattdessen eine Schleife über alle Spielstufen, deren Namen im Array `level_list` hinterlegt sind. Wir laufen durch das Arrays, bis wir den, laut Konvention am Ende des Arrays enthaltenen, `NULL`-Pointer erreichen.
Bauen Sie die Schleife, wie im Folgenden skizziert, ein:

```
// Play the game
// At the beginnung of the level, we still have a chance to win
game_state = WORM_GAME_ONGOING;
if (thegops.start_level_filename != NULL) {
      Code aus Teilaufgabe 6 ohne else-Zweig
} else {
        // Play all the levels in the level_list
        cur_level = 0;
        while (level_list[cur_level] != NULL && res_code == RES_OK &&
                game_state == WORM_GAME_ONGOING) {
                Schleifenrumpf: spiele aktuellen Level
                cur_level++;
        }
Nachbehandlung der Schleife: Wieso haben wir die Schleife verlassen?
```

[4]Hier besteht eine starke Ähnlichkeit zu den APA-Strukturen. Wir haben ebenfalls ein Array von Pointer auf Arrays, jedoch sind die Arrays im vorliegenden Fall alle statisch alloziert!

Hinweise zu den einzelnen Bausteinen: Schleifenrumpf: spiele aktuellen Level:

- Rufen Sie die Funktion `doLevel` auf und speichern Sie das Resultat in der Variablen `res_code`.
- Erhöhen Sie den Index `cur_level` für den nächsten Schleifendurchlauf

Nachbehandlung der Schleife: Wieso haben wir die Schleife verlassen?

Falls bei der Ausführung des Programms die **`while`**-Schleife verlassen wird und der Code der Nachbehandlung ausgeführt wird, müssen wir folgende Fälle unterscheiden:

1. Es ist ein Fehler im Programmablauf aufgetreten: `res_code != RES_OK`
   Zeigen Sie eine dementsprechende Nachricht an (`showDialog`)
2. Der Benutzer hat alle Level erfolgreich durchgespielt:
   `level_list[cur_level] == NULL && game_state == WORM_GAME_ONGOING`
   Eine derartige Leistung verdient natürlich ein besonderes Lob (`showDialog`)
3. Der Benutzer will das Spiel abbrechen: `game_state == WORM_GAME_QUIT`
   Drücken Sie Ihr Bedauern aus und sagen Sie ihm auf Wiedersehen (`showDialog`)
4. Der Benutzer hat einen Fahrfehler im letzten Level gemacht. (alle anderen Fälle)
   Trösten Sie ihn (`showDialog`)

Nach diesen Änderungen sollte Ihr Projekt wieder kompilierbar sein und fehlerfrei ausgeführt werden können. Spielen Sie nun Ihr Spiel mit allen Spielstufen durch und prüfen Sie die Steuerung der Schleife für die Spielstufen.

Aufgabe 8 (Dynamische Allokation des Arrays für die Wurmpositionen)

Bisher ist die maximale Länge des Benutzerwurms fest über die `CPP`-Konstante `WORM_LENGTH` vorgegeben. Dafür gibt es aber nun keinen Grund mehr, denn je nach Größe des Anzeigefensters und der Anzahl der ausgelegten Futterbrocken kann der Wurm eine unterschiedliche maximale Länge erreichen.
In der letzten Teilaufgabe dieses Aufgabenblatts werden wir deshalb das Array für die Abspeicherung der Wurmpositionen dynamisch allozieren. Als Vorgabe für diese Programmieraufgabe wählen wir beim Aufruf der Funktion `initializeWorm` in der Funktion `doLevel` den Wert
`(theboard.last_row + 1) * (theboard.last_col + 1)`
für die maximale Größe des Wurms. Größer kann der Wurm sicher nicht werden. Natürlich könnte man die maximale Größe auch noch von der Anzahl und Wertigkeit der Futterbrocken abhängig machen, aber derartige Varianten bleiben Ihnen für die individuelle Ausgestaltung des Spiels vorbehalten.
Unabhängig von diesen Erwägungen benötigen wir die dynamische Allokation der Speicherstruktur des Wurms für den späteren Einbau der Systemwürmer auf Blatt 11.
Beginnen Sie die Umgestaltung mit den folgenden Änderungen in der Header-Datei `worm_model.h`:

- Entfernen Sie die Definition der `CPP`-Konstanten `WORM_LENGTH`.
  Damit führt jede weitere Benutzung der Konstanten im Code zu einem Übersetzungsfehler.

- Ändern Sie die Definition der Komponente `wormpos` des Verbunds **`struct worm`** wie folgt:

  ```
  struct pos* wormpos;  // Array of x,y positions of ...
  ```

  Statt eines statisch allozierten Arrays hat der Verbund nun nur noch einen Pointer auf ein dynamisch alloziertes Array.

- Fügen Sie der Header-Datei zudem die folgenden beiden Funktionsdeklarationen hinzu.
  Die Motivation für diese beiden neuen Funktionen des Moduls `worm_model` erfolgt im Anschluss.

  ```
  extern void cleanupWorm(struct worm* aworm);
  extern void removeWorm(struct board* aboard, struct worm* aworm);
  ```

Da die Struktur **`struct worm`** nun kein statisch alloziertes Array mehr für die Wurmposition definiert, müssen wir die dynamische Allokation des Arrays an geeigneter Position durchführen. Der beste Ort hierfür ist die Funktion `initializeWorm` des Moduls `worm_model`. Die Funktion initializeWorm bekommt bereits die maximale Länge des Wurms als Parameter `len_max` übergeben. Wir müssen nur noch die gewünschte Anzahl an Speicher für das Array allozieren und die Adresse im Pointer `aworm->wormpos` speichern.
Öffnen Sie die Modul-Datei `worm_model.c` und fügen Sie den rot markierten Code hinzu.

```
// Initialize headindex
aworm->headindex = 0;   // Index pointing to head position is set to 0

// Initialize the array for element positions
// Allocate an array of the worms length
if ( (aworm->wormpos = malloc(sizeof(struct pos) * len_max)) == NULL) {
        showDialog("Abbruch: Zu wenig Speicher","Bitte eine Taste druecken");
        exit(RES_FAILED); // No memory -> direct exit
}
```

Da wir die Funktionen `malloc` und `showDialog` verwenden, müssen wir die entsprechenden Header-Dateien einbinden. Fügen Sie folgende `#include`-Direktiven zu Beginn der Modul-Datei `worm_model.c` ein.

```
#include <stdlib.h>
#include "messages.h"
```

Im Rahmen der Allokation von Speicher sollte man immer über die Lebensdauer des erzeugten Speicherobjekts auf dem Heap nachdenken. Die Funktion `initializeWorm`, die den Speicher für die Wurmpositionen alloziert, wird von der Funktion `doLevel` aufgerufen. In `doLevel` ist auch die Strukturvariable `userworm` definiert. Daher sollte auch die Funktion `doLevel` für die Freigabe des Speicherplatzes des dynamisch allozierten Speicherobjekts verantwortlich sein.
Im Sinne einer guten Modularisierung implementieren wir im Modul `worm_model` die Funktion
`void` `cleanupWorm(`**`struct worm`**`* aworm);`
die den durch `aworm->wormpos` referenzierten Speicher freigibt. Diese Funktion rufen wir dann zu einem geeigneten Zeitpunkt in der Funktion `doLevel` auf. Auf diese Weise sind wieder alle Funktionen, die Aspekte eines Wurms manipulieren, im Modul `worm_model` zusammengefasst (gekapselt)[5].
Fügen Sie die nachfolgende Definition der Funktion `cleanupWorm` der Modul-Datei `worm_model.c` hinzu. Ergänzen Sie den Code an der durch `@nnn` markierten Stelle:

[5] In einer objektorientierten Implementierung des Spiels wäre das Modul `worm_model` eine Klasse

```
void cleanupWorm(struct worm* aworm) {
        // free array of wormpos
        free( @030 );
}
```

Wir sind jetzt in der Lage, in der Funktion `doLevel` durch Aufruf der Funktion `initializeWorm` einen Wurm zu erzeugen (und Speicherplatz für die Komponente `wormpos` zu allozieren) und diesen Wurm durch Aufruf der Funktion `cleanupWorm` wieder zu zerstören. Nun wird in `doLevel` aber auch die Funktion `showWorm` aufgerufen, die die einzelnen Glieder des Wurms mittels `placeItem` auf dem Spielbrett registriert und auch im Puffer des Anzeigefenster an den entsprechenden Positionen speichert.
Vor der Zerstörung des Wurms mittels `cleanupWorm` sollten wir daher den Wurm auch sauber aus der Buchführung herausnehmen, das heißt wir sollten sowohl die Registrierung auf dem Spielbrett als auch die Speicherung im Puffer des Ausgabefensters zurücknehmen.
Derzeit ist diese Löschaktion für den Wurm noch nicht unbedingt notwendig, da der Wurm erst nach Beendigung einer Spielstufe am Ende der Funktion `doLevel` zerstört wird. Danach wird aber für die nächste Spielstufe eine neue Level-Datei eingelesen, die die Belegung des Spielbretts und auch den Anzeigepuffer neu initialisiert (siehe Funktion `initializeLevelFromFile`).
Derartige Abhängigkeiten sind immer unschön und stören meistens bei späteren Erweiterungen der Funktionalität. Das zeigt sich auch diesmal, denn im Rahmen der Einführung der Systemwürmer auf Blatt 11 müssen wir sowohl den Benutzerwurm als auch Systemwürmer vom Spielfeld verschwinden lassen können, auch wenn die aktuelle Spielstufe noch nicht zu Ende gespielt ist. Wir können uns dann also nicht mehr darauf verlassen, dass die Funktion `initializeLevelFromFile` unsere Hinterlassenschaften bereinigt.
Aus diesem Grund gehen wir in Vorlage und implementieren im Model `worm_model` die Funktion
`void removeWorm(struct board* aboard, struct worm* aworm);`
die den via Pointer `aworm` übergebenen Wurm vom Spielbrett `aboard` und aus dem Anzeigepuffer eliminiert.
Die Funktion `removeWorm` ist, vom Ablauf her, wie die Funktion `isInUseByWorm` der Version `Worm050` aufgebaut. Implementieren Sie die Funktion nach folgendem Muster, wobei Sie wieder die durch `@nnn` gekennzeichneten Stellen vervollständigen müssen.

```
// Remove a worm from the board and clean the display
void removeWorm(struct board* aboard, struct worm* aworm) {
        int i;
        i = aworm->headindex;
        do {
                placeItem( @040 );
                // Advance index; go round after aworm->cur_lastindex
                @041
        } while ( @042 );
}
```

Jetzt fehlen nur noch die entsprechenden Aufrufe in der Funktion `doLevel` für die in dieser Teilaufgabe eingebrachten Neuerungen. Öffnen Sie die Datei `worm.c` und nehmen Sie folgende letzte Änderungen vor.
Ändern Sie den Aufruf der Funktion `initializeWorm`:

```
res_code = initializeWorm(&userworm,(theboard.last_row+1)*(theboard.last_col+1),
                    WORM_INITIAL_LENGTH, bottomLeft, WORM_RIGHT, COLP_USER_WORM);
```

Rufen Sie am Ende der Funktion `doLevel`, kurz bevor die Funktion verlassen wird, zuerst die

Funktion `removeWorm` auf, damit der Wurm vom Spielfeld gelöscht wird und danach die Funktion `cleanupWorm` auf, die den dynamisch allozierten Speicher wieder freigibt:

```
// Normal exit point

// Remove the worm from display and board
removeWorm(&theboard, &userworm);
// Because initializeWorm allocates memory
cleanupWorm(&userworm);

cleanupBoard(&theboard);
return res_code;
```

Endkontrolle

Ihr Projekt `Worm090` 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.
Hiermit haben Sie alle Teilaufgaben dieses Aufgabenblatts 10 bearbeitet.
Zum Abschluss speichern Sie die Quellen Ihrer finalen lauffähigen Version im Repository.
Zum Beispiel so:

```
$ git status
$ git add .
$ git commit -m "Worm090 final"
```

Schlußbemerkung

Im Zuge dieses Aufgabenblatts haben wir die folgende Funktionalität zu unserem Spiel hinzugefügt:

- Einlesen von Spielfeldbeschreibungen aus Level-Dateien.
- Der Name des zu spielenden Levels kann auf der Kommandozeile angegeben werden.
- Die Funktion `playGame` steuert in einer Schleife die Ausführung einer Sequenz von Spielstufen. Für jede Spielstufe wird eine Beschreibung des Spielfelds aus einer Level-Datei eingelesen. Die Liste der Dateinamen ist in der Funktion `playGame` fest verdrahtet.
- Das Array für die Speicherung der Positionen der einzelnen Wurmglieder wird für jede Spielstufe dynamisch alloziert und auch wieder freigegeben.
- Der Wurm kann vom Spielfeld und auch aus dem Puffer der Anzeige gelöscht werden.

Das letzte Aufgabenblatt (Blatt 11) des Praktikums erweitert Ihr Spiel um automatisch bewegte Systemwürmer. **Die Bearbeitung von Blatt 11 ist optional**, zeigt Ihnen jedoch, wie Sie aufgrund eines guten Software-Designs ganz einfach statt nur einem Wurm eine beliebige Anzahl von Würmern verwalten können.
Ungeachtet dessen, ob Sie das Blatt 11 bearbeiten, wird Ihnen, nach erfolgreicher Vorführung der

anhand der Aufgaben von Blatt 09 und Blatt 10 entwickelten Funktionalität, von Ihrem Betreuer das letzte noch fehlende Testat für das Praktikum Grundlagen der Programmierung 1 ausgestellt. Im Rahmen des Praktikums haben Sie, unter Anleitung, das Spiel Worm entwickelt. In der Gesamtbetrachtung handelt es sich hierbei um ein nicht-triviales Software-Projekt, das Sie nun erfolgreich in der Programmiersprache C, unter Benutzung einer GNU-Entwicklungsumgebung, umgesetzt haben. Auf den folgenden Seiten finden Sie Hinweise über mögliche individuelle Erweiterungen Ihres Spiels.

Herzlichen Glückwunsch und viel Spaß mit Ihrem Spiel Worm!

1 Einbau eines Zufallsgenerators

Dieser Abschnitt erklärt kurz, wie Sie in Ihrem Spiel einen Generator für Pseudo-Zufallszahlen einbinden können. Die C-Standardbibliothek stellt im Modul `stdlib` Funktionen zur Generierung von Pseudo-Zufallszahlen bereit:

```
#include <stdlib.h>
int rand(void);
void srand(unsigned int seed);
```

Vor der Erzeugung von Zufallszahlen müssen Sie den Generator zuerst mittels der Funktion srand initialisieren. Damit bei jedem Start des Spiels eine andere Folge von Pseudo-Zufallsszahlen generiert wird, übergeben Sie der Funktion `srand` einen Wert, der sich bei jedem Lauf Ihres Programms verändert.
Ein guter Kandidat für einen solchen garantiert wechselnden Wert ist das Resultat des Aufrufs der Funktion time, die ebenfalls in der C-Standardbibliothek enthalten ist:

```
#include <time.h>
time_t time(time_t *t);
```

Der Aufruf `time(NULL)` liefert die seit dem 1. Januar 1970 um 00:00 vergangene Anzahl von Sekunden. Dieser Wert ändert sich derzeit noch immer ständig ;-)

Durch Hinzufügen der folgenden Anweisung in der Funktion `main` der Datei `worm.c` können Sie den Zufallsgenerator der C-Standardbibliothek initialisieren:

```
// Initialize random generator
srand ( time(NULL) );
```

Natürlich müssen Sie dann auch die folgenden `#include`-Direktiven am Beginn Ihrer Datei `worm.c` einfügen:

```
#include <stdlib.h>
#include <time.h>
```

Nun können Sie überall in Ihrem Programm Zufallszahlen generieren. Die Funktion `rand` liefert bei jedem Aufruf eine zufällige ganze Zahl im Bereich `[0, RAND_MAX]`. Der Wert der Konstante `RAND_MAX` hängt dabei von der Plattform (Windows, Linux), dem C-Compiler und dem Prozessor

Ihres Computers ab.
Auf einem 32 Bit Linux hat die Konstante `RAND_MAX` den Wert 2147483647. Das können Sie leicht durch folgendes Programm überprüfen:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
        printf("RAND_MAX=%d\n",RAND_MAX);
        return 0;
}
```

Um eine ganzzahlige positive Zufallszahlen `z` im Bereich `[i,j]` zu erzeugen, implementieren Sie Code nach folgendem Muster:

```
int  i, j, z;
z  =  i + (rand() % (j - i + 1));
```

Um eine positive Fließpunktzahl `f` im Bereich `[a,b]` zu erzeugen, implementieren Sie Code nach folgendem Muster:

```
double a,b,f;
f  =  a  + ( 1.0 * rand()  /  RAND_MAX ) * (b - a);
```

Individuelle Erweiterungen

Im Folgenden sind ein paar Ideen für individuelle Erweiterungen des Spiels aufgeführt. Deren Umsetzung wird Ihnen nach Absolvierung des Praktikums keine nennenswerten Probleme mehr bereiten.

1. Der Wurm wächst sofort sichtbar nach Fressen eines Futterbrockens.
2. Die Lage, Wertigkeit und Anzahl von Futterbrocken wird nicht nur in den Level-Dateien vorgegeben, sondern zusätzlich durch den Einsatz eines Zufallsgenerators bestimmt. Ein Mischverfahren soll gewährleisten, dass weiterhin Level-Dateien korrekt verarbeitet werden, die Futterbrocken festlegen.
3. Die Anzahl der Futterbrocken für die einzelnen Wertigkeiten kann auf der Kommandozeile durch die Optionen `[-f1 n] [-f2 m] [-f3 k]` angegeben werden.
4. Der Wurm ändert je nach gefressenem Futterbrocken die Farbe seiner Glieder. Dabei nehmen genau so viele Glieder die Farbe des Futterbrockens an, wie er aufgrund der Konsumierung des Futterbrockens wächst.
5. Der Wurm schrumpft auch wieder, wenn er zu lange keinen Futterbrocken gefressen hat. Die Zeit bis zum Beginn des Schrumpfprozesses (Anzahl von ausgeführten Einzelschritten) sowie die Anzahl der zu löschenden Wurmglieder können als Optionen auf der Kommandozeile spezifiziert werden.
6. Statt nur einer Level-Datei kann man eine Liste aller zu spielenden Level-Dateien auf der Kommandozeile angeben.
7. Pro Level stehen mehrere Level-Dateien zu Auswahl. Aus der Menge der Kandidaten wird zufällig für jeden Level eine Level-Datei ausgewählt.

8. Vor Spielbeginn wird eine Animation (ASCII-Art) als Intro unter Benutzung eines Zufallsgenerators eingeblendet.

9. Die Glieder des Wurms werden unter Benutzung von UTF-8 Zeichen oder durch Zeichen aus dem Alternative Character Set (ACS) von Curses interessanter gestaltet.

10. Ein Funktion zur Generierung von zufälligen Level-Dateien wird implementiert. Dabei werden bestimmte Grundregeln eingehalten, um nur spielbare Level-Beschreibungen zu erzeugen.

11. u.s.w

Die Grenzen Ihrer Kreativität bestimmen Sie selbst !!!11Einself