Technische Hochschule Ingolstadt
Fakultät für Informatik

Praktikum zu „Grundlagen der Programmierung“

Blatt 11

Lernziele: Das große friwillige Finale
Erforderliche Kenntnisse: Inhalte des ersten Semesters
Voraussetzungen:

- 1. Vollständige Bearbeitung des letzten Blattes 10 (`Worm090`).

Übersicht

Auf diesem Blatt, dessen Bearbeitung freiwillig ist, werden wir zusätzlich zum Wurm des Benutzers eine beliebige Anzahl an automatisch bewegten Würmern, sogenannte Systemwürmer, zu unserem Spiel hinzufügen. Die Anzahl ist nur durch die Größe des Anzeigefensters eingeschränkt, denn von jedem Wurm muss mindestens das Kopfelement auf dem Spielbrett Platz finden.

Die Einführung einer beliebigen Anzahl von Systemwürmern wird bereits von unseren Datenstrukturen unterstützt, die wir bis einschließlich Blatt 10 aufgebaut haben. Wir müssen jetzt nur noch die entsprechende Logik dafür in unser Spiel einbauen.

Die Erweiterungen zerfallen in zwei Bereiche. Auf der einen Seite brauchen wir Funktionen, die

- einen einzelnen Systemwurm erzeugen inklusive zufälliger Farbe und initialer Laufrichtung,
- einen einzelnen Systemwurm vom Spielfeld nehmen und den Speicher freigeben,
- einem Wurm eine beliebige neue Laufrichtung geben (an Stelle des Benutzers),
- die eben genannten Funktionen für alle Systemwürmer ausführen.

Auf der anderen Seite müssen wir die neuen Funktionen für die Systemwürmer im Code unseres Spiels an passender Stelle aufrufen, damit die Verwendung der Systemwürmer in das bestehende Spiel integriert wird. Konkret werden wir die Funktionen für die Systemwürmer alle in der Funktion `doLevel` aufrufen.

In der Nachbertachtung stellt sich heraus, dass die Integration der neuen Funktionalität in die bisherige Ablaufsteuerung mehr Aufwand bereitet als die Implementierung der neuen Funktionen für die Systemwürmer. Insbesondere die korrekte Behandlung neuer Fehler, die von Systemwürmern verursacht werden können, erweist sich als aufwändig.

Aus diesem Grund werden auf diesem Aufgabenblatt der Code für die Integration der Systemwürmer und die Fehlerbehandlung weitgehend vorgegeben[1]. Die Implementierung der neuen Funktionen für die Systemwürmer wird dagegen nur grob skizziert. An dieser Stelle können Sie dann Ihre

[1] Das soll und kann Sie natürlich nicht daran hindern, ganz eigene Wege zu gehen. Es wird nur eine mögliche Implementierung präsentiert.

gerade frisch erworbenen Programmierkünste auf die Probe stellen und jede Menge individueller Varianten programmieren.

Aufgabe 1 (Kopieren, Auspacken und Umbenennen des Templates)

Im Moodle-Kursraum finden Sie im Verzeichnis `Praktikum` die Datei `Worm100Template.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 Worm100Template.zip`
Entpacken Sie die Zip-Datei durch den nachfolgenden Befehl. Dadurch wird das Verzeichnis `Worm100Template` mit einigen Dateien darin angelegt.
`$ unzip Worm100Template.zip`
Listen Sie den Inhalt des neuen Unterverzeichnisses.
`$ ls -la Worm100Template`
Benennen Sie das Verzeichnis `Worm100Template` um in `Worm100`. `$ mv Worm100Template Worm100`
Listen Sie den Inhalt des umbenannten Unterverzeichnisses.
`$ ls -la Worm100`
Listen Sie den Inhalt des aktuellen Verzeichnisses.
`$ ls -la`
Löschen Sie das Zip-Archiv.
`$ rm -f Worm100Template.zip`
Führen Sie den nachfolgenden Kopierbefehl[2] aus. Dadurch werden fast alle Modul-Dateien (`*.c`) nebst zugehörigen Header-Dateien aus dem Verzeichnis `Worm090` in das neue Verzeichnis `Worm100` übernommen (kopiert).
Damit schaffen Sie die Ausgangssituation für die Neuerungen in der Version `Worm100`.

```
$ cd Worm090
$ cp board_model.* prep.* worm.* worm_model.* ../Worm100
$ cd Worm100
```

Aufgabe 2 (Test der Ausgangssituation)

Öffnen Sie eine Shell und wechseln Sie in dieser Shell ins Verzeichnis
`~/GdP1/Praktikum/Code/Worm100` Eine Auflistung der Dateien in diesem Verzeichnis sollte folgenden Inhalt anzeigen:

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

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

Die blau dargestellten Dateien haben Sie zum Schluss der letzten Aufgabe aus dem Verzeichnis der vorherigen Version `Worm090` kopiert, die anderen Dateien sind durch das Auspacken des Templates `Worm100Template.zip` entstanden.
Stellen Sie sicher, dass Ihr Projekt im Verzeichnis `Worm100` kompilierbar ist. Führen Sie dazu folgenden Befehl im Verzeichnis `Worm100` 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.

Hinweis: nun wäre ein guter Zeitpunkt, um diese initiale Variante im Repository zu speichern.

Vorgabe aus dem Template

Im Template `Worm100Template.zip` sind mehrere Dateien enthalten, die im Folgenden kurz besprochen werden.

messages.*: enthält eine kleine Änderung relativ zur Vorversion, die das Löschen der gesamten Message Area als Funktion `clearMessageArea` anbietet.

options.*: es wurden folgende neuen Optionen eingeführt (siehe `usage.txt`)

- Angabe der gewünschten Anzahl an Systemwürmer `[-A n]`
- Einblenden der Systemwürmer gleich am Anfang des Spiels `[-a]`
- Ausblenden des Benutzerwurms gleich am Anfang des Spiels `[-u]`

sysworm.*: Hierbei handelt es sich um ein neues Modul, das alle Funktionen enthält, die zur Manipulation von Systemwürmern benötigt werden. Die Header-Datei `sysworm.h` enthält bereits alle Funktionsdeklarationen. Die Modul-Datei `sysworm.c` enthält in der Version, die aus dem Template kommt, nur leere Funktionsrümpfe. Dort müssen Sie dann tätig werden (siehe Teilaufgabe 4).

worm.c.inc: Diese Datei enthält den Code, der die Systemwürmer ansteuert. Diesen Code müssen Sie an geeigneter Stelle in Ihre Funktion `doLevel` einkopieren (siehe Teilaufgabe 3).

Makefile: Das `Makefile` wurde geringfügig erweitert, so dass mit dem Target `dist` eine eigenständige Distribution im Unterverzeichnis `dist` erzeugt werden kann. Die ausführbare Datei `worm` wird dabei statisch mit allen Bibliotheken gebunden, so dass das Programm auf jedem beliebigen Rechner mit passendem Betriebssystem lauffähig ist.

***.level.*:** Alle Dateien mit diesem Muster enthalten Beschreibungen von Spielfeldern (Level). Es handelt sich hierbei um die bereits bekannten Dateien aus vorherigen Versionen, die Sie problemlos durch Ihre eigenen Level-Beschreibungen ersetzen können.

Aufgabe 3 (Erste Erweiterungen von `worm.h` und `worm.c`)

Erweiterung `worm.h`:

In einem ersten Schritt erweitern wir die Datei `worm.h` um eine neue Farbe und mehrere Codes für die Systemwürmer. Führen Sie folgende Erweiterungen in `worm.h` durch:

```
// Numbers for color pairs used by curses macro COLOR_PAIR
enum ColorPairs {
  COLP_USER_WORM = 1, // Must be color pair with code 1
  COLP_FREE_CELL,     // Must be color pair with code 2
  COLP_SYSWORM,
  COLP_FOOD_1,
  COLP_FOOD_2,
  COLP_FOOD_3,
  COLP_BARRIER
};
#define FIRST_AUTO_COLP   COLP_SYSWORM
#define LAST_AUTO_COLP    COLP_BARRIER
```

Die Definitionen `FIRST_AUTO_COLP` und `LAST_AUTO_COLP` dienen dazu, das Zielintervall für den Zufallsgenerator zu beschränken, wenn wir später eine Farbe für einen Systemwurm auswählen. Der Wurm soll natürlich nicht so aussehen wie der Benutzerwurm (`COLP_USER_WORM`) und auch nicht unsichtbar sein.
Danach fügen wir noch ein paar Fehlercodes hinzu, die im weiteren Verlauf im Programm benutzt werden. Erweitern Sie **`enum`** `ResCodes` um:

```
RES_NO_FREE_CELL_ERROR,
RES_NO_FREE_DIRECTION_ERROR,
RES_NO_MEMORY,
RES_USERWORM_ERROR,
RES_SYSWORM_ERROR,
```

Erweitern Sie **`enum`** `GameStates` um:

```
WORM_USERWORM_ERROR,  // Problem with userworm
WORM_SYSWORM_ERROR,   // Problem with sysworms
```

`#include`-Direktive für `sysworm` in `worm.c`:

Im zweiten Schritt erweitern wir die Datei `worm.c` um eine neue `#include`-Direktive, mit der wir die Bezeichner des neuen Moduls `sysworm` bekannt machen. Fügen Sie die Direktive wie skizziert am Ende der bisherigen `#include`-Direktiven ein.

```
#include "worm_model.h"
#include "board_model.h"
#include "sysworm.h"
```

Desweiteren sollten Sie noch das neue Farbpaar in der Funktion `initializeColors` initialisieren. Etwa so: `init_pair(COLP_SYSWORM, COLOR_BLUE, COLOR_BLACK);`

Systemwürmer in `doLevel` erzeugen und anzeigen:

Wie eingangs erwähnt, gehen wir davon aus, dass wir im Modul `sysworm` alle benötigten Funktionen für Systemwürmer implementieren werden. Zu diesen Funktionen zählen auch die folgenden beiden, die wir hier im Aufrufkontext von `doLevel` zeigen:

```
initializeSysWormsArray(somegops,&sysworms);
```

Der Zeiger `sysworms` soll auf ein dynamisch alloziertes Array von Strukturen des Typs **`struct worm`** zeigen, in denen wir die Systemwürmer speichern werden. Die Funktion `initializeSysWormsArray` alloziert Speicher für dieses Arrays und weist die Basisadresse an den Pointer `sysworms` zu, der deswegen per Adresse übergeben wird. Die gewünschte Anzahl der zu erstellenden Systemwürmer ist in `somegops->sysworms_nr` hinterlegt.

```
enableSysworms(somegops,&theboard,sysworms);
```

Die Funktion `enableSysworms` wird für jeden Wurm im Array `sysworms` eine zufällige Farbe wählen und dann eine Hilfsfunktion aufrufen, die für diesen einzelnen Wurm Anfangsposition und Richtung festlegt und Speicher für den Ringpuffer des Wurms alloziert. Danach wird der Wurm auch gleich noch angezeigt. Der Aufruf von `enableSysworms` wird also mit einem Schlag alle Systemwürmer erzeugen und anzeigen.
Den nachfolgenden Code müssen Sie am Anfang der Funktion `doLevel` einfügen. Zuerst müssen Sie die Zeigervariable `sysworms` definieren:

```
struct worm userworm;  // Currently we just use one user worm in the game
struct worm* sysworms; // Pointer to dynamically allocated array of system worms
```

Etwas weiter unten, am besten nach der Initialisierung des Benutzerwurms rufen Sie dann die beiden eben skizzierten Funktionen `initializeSysWormsArray` und `enableSysworms` mit entsprechender Fehlerbehandlung auf (siehe unten). Damit sind dann die Systemwürmer initialisiert und die Hauptschleife in `doLevel` kann beginnen.

```
// Allocate memory for the array of sysworms
if (somegops->sysworms_nr > 0) {
  res_code = initializeSysWormsArray(somegops,&sysworms);
  if ( res_code != RES_OK) {
    return res_code;
  }
}
// Initialize and show sysworms as well, if they are active.
// Assertion: if worms are to be active there exists at least one worm
if ( somegops->activate_sysworms ) {
  enum ResCodes res_code = enableSysworms(somegops,&theboard,sysworms);
  if ( res_code != RES_OK) {
    *agame_state = WORM_SYSWORM_ERROR;
    return res_code;
  }
}
```

Systemwürmer in der Hauptschleife von `doLevel` bewegen:

Nun kommen wir zum kompliziertesten Teil des Einbaus der Systemwürmer. In der Schleife von `doLevel` steht bisher in etwa folgender Code-Block:

```
readUserInput(&userworm, agame_state);
if ( *agame_state == WORM_GAME_QUIT ) {
  end_level_loop = true;
```

```
  continue; // Go to beginning of the loop's block ...
}
cleanWormTail(&theboard, &userworm);
moveWorm(&theboard, &userworm, agame_state);
if ( *agame_state != WORM_GAME_ONGOING ) {
  end_level_loop = true;
  continue; // Go to beginning of the loop's block ...
}
showWorm(&theboard, &userworm);
```

Damit wird der User-Wurm einen Schritt bewegt. Vereinfacht dargestellt müssen wir nun im Anschluss an diesen Block einfach für jeden der Systemwürmer das gleiche machen, nämlich ihn einen Schritt bewegen. Bei den Systemwürmern gibt aber nicht der Benutzer durch Tastatureingabe die Richtung vor, sondern wir berechnen für jeden Systemwurm per Zufallsgenerator bei jedem Schritt eine neue Richtung. Die hierzu benötigten Funktionen werden Sie im Modul `sysworm` implementieren, aber in `doLevel` müssen diese Funktionen aufgerufen werden.
Da die Logik für diesen Block nicht ganz einfach ist, wird der Block komplett in der Datei `worm.c.inc` vorgegeben. Wir besprechen im Anschluss jetzt die einzelnen Zeilen dieses Blocks, wobei die Zeilennummern sich auf die Datei worm.c.inc beziehen und nicht auf worm.c! Sie müssen im Anschluss an diese Besprechung den Code der Datei `worm.c.inc` an der oben angedeuteten Stelle in die Funktion `doLevel` einfügen.

Besprechung des Inhalts der Datei **worm.c.inc**

```
1  // START process sysworms if they are active
2  if ( somegops->activate_sysworms ) {
3    int w;  // Index for sysworms
4    enum GameStates worm_state = WORM_GAME_ONGOING ;
5    // For all sysworms
6    for (w = 0; w < somegops->sysworms_nr && worm_state == WORM_GAME_ONGOING; w++) {
7      // Compute a new direction for the sysworm
8      res_code = computeDirection(&theboard, &sysworms[w]);
9      if (res_code != RES_OK) {
10       enum ColorPairs color;
11       // We are not able to find a new direction for that sysworm
12       // Re-initialize that sysworm
13       removeWorm(&theboard, &sysworms[w]);
14       // Because initializeWorm allocates memory
15       cleanupWorm(&sysworms[w]);
16       // Randomly choose a color from all colors available for . . .
17       color = FIRST_AUTO_COLP + (rand() % (LAST_AUTO_COLP - FIRST_AUTO_COLP + 1));
18       // Create that worm again
19       res_code = initializeAndShowAutoWorm(&theboard,&sysworms[w],
20          WORM_INITIAL_SYSWORM_LENGTH,color);
21       if (res_code != RES_OK) {
22         // We are unable to place a new sysworm onto the board.
23         // Cancel loops and propagate state
24         end_level_loop = true; // Signal termination of level loop
25         *agame_state = WORM_SYSWORM_ERROR; // Propagate error
26          break; // Terminates loop forall sysworms
27       }
28     } else {
29       // Clean the tail of that sysworm
30       cleanWormTail(&theboard, &sysworms[w]);
31       // Now move the worm for one step
32       // computeDirection succeeded -> moveWorm will succeed
33       moveWorm(&theboard, &sysworms[w], &worm_state);
34       // Show the sysworm at its new position
35       showWorm(&theboard, &sysworms[w]);
36      }
37   }
```

```
38    // Propagate state; by construction we should always have here
39    //  worm_state == WORM_GAME_ONGOING
40    *agame_state = worm_state;
41  }
42  // END process sysworm(s)
```

Zeile 2-42: Code nur ausführen, falls Systemwürmer aktiv sind.

Zeile 6-38: Schleife über alle Systemwürmer.

Zeile 8: Hier berechnen wir für den Systemwurm mit Index `w` eine neue Richtung. Die Berechnung kann fehlschlagen, wenn es keine Bewegungsrichtung gibt, in der der Wurm den nächsten Schritt ohne Kollision durchführen kann.

Zeilen 30-37: Falls wir in Zeile 8 eine neue Richtung berechnen konnten, dann wird der Block in den Zeilen 30 bis 37 ausgeführt. Das ist der einfache Fall, der analog zur Behandlung des Benutzerwurms ausgeführt wird.

Zeilen 10-28: Falls wir keine neue Richtung für den Systemwurm mit Index `w` berechnen konnten, nehmen wird den Systemwurm von Spielfeld, geben den von ihm belegten Speicher zurück und erzeugen den Wurm ganz neu mit neuer Farbe und Anfangsposition. Sollte es keinen freien Platz mehr für das Kopfelement geben, dann signalisieren wir das über den Code `WORM_SYSWORM_ERROR` und leiten die Beendigung der Hauptschleife in `doLevel` ein.

Am Ende von `doLevel` abräumen und Speicher freigeben:

Ganz am Ende der Funktion `doLevel` müssen wir noch, analog zur Behandlung des Benutzerwurms, die angezeigten Systemwürmer vom Spielbrett nehmen und den von ihnen allozierten Speicher freigeben. Beim Start des nächsten Levels werden sie dann wieder erzeugt und auf dem Spielfeld angezeigt.
Fügen Sie den nachfolgend Code nach den Aufräumarbeiten für den Benutzerwurms am Ende der Funktion `doLevel` ein.

```
// If there are any sysworms active we need to do a cleanup
if ( somegops->activate_sysworms ) {
  // Disable sysworms
  disableSysworms(somegops,&theboard,sysworms);
}
if ( somegops->sysworms_nr > 0) {
  cleanupSyswormsArray(sysworms);  // Free memory allocated for
  // array storing the sysworms
}
```

Die Funktion `disableSysworms` löscht alle sich auf dem Spielbrett befindenden Systemwürmer und gibt den von ihnen allozierten Speicher frei (Pendant zu `enableSysworms`). Die Funktion `cleanupSyswormsArray` gibt den Speicher des Arrays frei, auf den der Pointer `sysworms` zeigt (Pendant zu `initializeSyswormsArray`).

Aufgabe 4 (Funktionen in `sysworm.c`)

In dieser Aufgabe müssen Sie die einzelnen Funktionen des Moduls `sysworm` implementieren.
Im Unterschied zu früher erhalten Sie aber kaum noch Vorgaben in den Code-Schablonen. Sie müssen also die Funktionen weitgehend selbst implementieren. Wir besprechen hier lediglich grob, was die Funktionen leisten sollen, damit die in `doLevel` eingefügten Code-Zeilen wie gedacht funktionieren.

initializeSyswormsArray:

```
enum ResCodes initializeSyswormsArray(struct game_options* somegops,
  struct worm** ptrToSyswormsArray) {
  enum ResCodes res_code = RES_OK;
  // @nnn
  return res_code;
}
```

Die Funktion soll ein Array mit Elementen vom Typ `struct worm` dynamisch allozieren und die Basisadresse dieses Arrays dem per Adresse übergebenen Pointer `ptrToSyswormsArray` zuweisen. Die Anzahl der im Arrays benötigten Elemente können Sie der Komponente `somegops->sysworms_nr` entnehmen.

Stellen Sie sicher, dass Sie den Typ des Parameters `struct worm** ptrToSyswormsArray` verstehen.
Im Array sind Elemente des Typs `struct worm` gespeichert (für jeden Systemwurm eines), die Basisadresse des Arrays selbst hat den Typ `struct worm*`, und da Sie diese Basisadresse in einer als Parameter übergebenen Variablen speichern sollen, muss dieser Parameter per Adresse übergeben werden. Sonst stellt sich beim Aufrufer der gewünschte Seiteneffekt nicht ein. Daher der Typ `struct worm**`.

cleanupSyswormsArray:

```
void cleanupSyswormsArray(struct worm* ptrToSyswormsArray) {
  // @nnn
}
```

Diese Funktion soll den durch `initializeSyswormsArray` allozierten Speicher des Arrays wieder freigeben. Beim Aufruf wird der Pointer übergeben, der von `initializeSyswormsArray` als Ergebnis geliefert wurde.

initializeAndShowAutoWorm:

```
enum ResCodes
initializeAndShowAutoWorm(struct board* aboard, struct worm* aworm,
  int len, enum ColorPairs color) {
  enum ResCodes res_code = RES_OK;
  // Schritt 1: suche freie Zelle auf dem Spielbrett: findFreeCell
  // Schritt 2: initialisiere diesen Wurm: initializeWorm
  // Schritt 3: den Wurm anzeigen: showWorm
  // Schritt 4: eine zufällige Richtung berechnen: computeDirection
  return res_code;
}
```

Die Funktion `initializeAndShowAutoWorm` wird von der Funktion `enableSysworms` (siehe unten) für jeden sich im Array der Systemwürmer befindenden Systemwurm aufgerufen. Die einzelnen durch die Funktion auszuführenden Schritte sind im obigen Template angedeutet.
Die Funktion `findFreeCell` werden Sie im Rahmen der Aufgabe 5 im Modul `board_model` implementieren. Sie wird wie folgt aufgerufen (Details in Aufgabe 5):

```
// Find a place where to put the head of the worm
res_code = findFreeCell(aboard, &headpos);
```

Beachten Sie, dass nach der Initialisierung des Wurms von diesem nur das Kopfelement sichtbar ist, da der Ringpuffer des Wurms ja ebenfalls initialisiert wird. Die berechnete Laufrichtung des Wurms ist erst im nächsten Schritt von Bedeutung.

enableSysworms:

```
enum ResCodes enableSysworms(struct game_options* somegops,
  struct board* aboard, struct worm sysworms[]) {
  enum ResCodes res_code = RES_OK;
  // For all sysworms
  int w;
  for (w = 0; w < somegops->sysworms_nr; w++) {
    // @nnn
  }
  return res_code;
}
```

Die Funktion `enableSysworms` führt eine Schleife über alle Systemwürmer aus und berechnet zuerst für jeden zu erzeugenden Wurm eine Farbe per Zufallsgenerator. Dabei kommen die `CPP`-Konstanten `LAST_AUTO_COLP` und `FIRST_AUTO_COLP` zum Einsatz. Dann wird dieser Wurm per Aufruf der Funktion `initializeAndShowAutoWorm` initialisiert und angezeigt (siehe ebenfalls Datei `worm.c.inc`).

disableSysworms:

```
void disableSysworms(struct game_options* somegops, struct board* aboard,
  struct worm sysworms[]) {
  // Schritt 1: Wurm vom Spielfeld löschen: removeWorm
  // Schritt 2: Speicher von Wurm freigeben: cleanupWorm
}
```

Die Funktion `disableSysworms` führt eine Schleife über alle Systemwürmer aus, löscht jeden Wurm und gibt den für ihn allozierten Speicher zurück.

isGoodHeading:

```
bool isGoodHeading(struct board* aboard, struct pos next_pos) {
  // @nnn; replace the following line by some useful code
  return false;
}
```

Die Funktion `isGoodHeading` ist eine von `computeDirection` genutzte Hilfsfunktion, die prüft ob an der Position `next_pos` ein freier Platz oder ein Futterbrocken liegt (*good heading*). Desweiteren wird geprüft, ob die Position sich überhaupt auf dem Spielbrett befindet. Andernfalls ist es keine gute Idee, den Systemwurm auf diese Position zu bewegen.

computeDirection:

```
enum ResCodes computeDirection(struct board* aboard, struct worm* aworm) {
  enum ResCodes res_code = RES_OK;
  // @nnn
  return res_code;
}
```

Die Funktion `computeDirection` berechnet, ausgehend von der aktuellen Position des Systemwurms, eine neue Laufrichtung für den Wurm und speichert diese in der Wurmstruktur. Dabei werden unter Verwendung der Hilfsfunktion `isGoodHeading` zunächst alle Kandidaten für eine mögliche Laufrichtung ermittelt, so dass es zu keinen unerwünschten Kollisionen im nächsten Schritt kommt. Unter diesen Kandidaten wird dann eine Laufrichtung per Zufallsgenerator ermittelt.

Tip: durch Abziehen der Koordinaten der neuen Position von den Koordinaten der aktuellen Position

erhält man die Delta-Werte für die Laufrichtung (`dx`, `dy`).

Damit die Bewegung der Systemwürmer nicht zu hektisch wirkt, können Sie zum Beispiel in 80% der Fälle die aktuelle Richtung beibehalten und nur in 20% der Fälle unter den anderen Alternative eine auswählen.
Falls Sie viel Energie in die Auswahl einer neuen Richtung stecken möchten, untersuchen Sie nicht nur die unmittelbare Umgebung des Wurmkopfs (ein Zeichen entfernt vom Kopf) sondern erforschen die Umgebung über eine größere Distanz. So können Sie Ihre Würmer schlauer machen, so dass sie Futterbrocken finden, wenn diese in der Nähe sind.
Falls Ihnen die gerade gegeben Skizzierung der Funktion im Modul `sysworm` noch nicht ausreicht, lesen Sie noch einmal die Ausführungen in der Aufgabe 3. Dort werden, bis auf `isGoodHeading`, alle Funktionen im Aufrufkontext gezeigt.

Aufgabe 5 (Funktionen in `board_model.c`)

Im Modul `board_model` fallen nur wenige Änderungen an. Im Wesentlichen wird nur die neue Funktion `findFreeCell` implementiert, die unter allen freien Zellen auf dem Spielbrett eine per Zufall auswählt. Die Position dieser Zelle wird dann im per Adresse übergebenen Parameter `apos` gespeichert.

Fügen Sie in der Header-Datei `board_model.h` die Signatur der neuen Funktion hinzu:

```
extern enum ResCodes findFreeCell(struct board* aboard, struct pos* apos);
```

Bevor Sie nun in der Modul-Datei `board_model.c` die Funktion `findFreeCell` implementieren, müssen Sie zuvor noch eine globale Variable **`struct pos*`** `free_list` in der Modul-Datei definieren. Die globale Variable soll nur im Modul `board_model` sichtbar sein, daher deklarieren wir sie als **`static`**. Damit kann Sie in anderen Modulen nicht mehr benutzt werden.

```
// A global variable that is only visible in this modul due to modifier 'static'
// Pointer to free list of cells.
// The memory that is pointed to by that variable is allocated
// in initializeBoard and the memory is freed in cleanupBoard
static struct pos* free_list;
```

Dieser Pointer soll auf ein Array von Positionen zeigen, die alle frei sind. Da die Größe des Anzeigefensters erst zur Laufzeit feststeht, allozieren wir das Arrays dynamisch.
Den Code für die Allokation des Arrays fügen Sie bitte am Ende der Funktion `initializeBoard` etwa wie folgt ein:

```
// Allocate an array of positions. We encode the free_list as an array!
// If the board is completely empty the free_list contains all positions
// of the board.
if ((free_list = malloc(sizeof(struct pos) * ((aboard->last_row+1) *
(aboard->last_col +1)))) == NULL) {
showDialog("Abbruch: Zu wenig Speicher","Bitte eine Taste druecken");
return RES_NO_MEMORY;
}
return RES_OK;
```

Damit steht Speicher für das Array der freien Zellen bereit, das über die globale Pointervariable `free_list` erreichbar ist. Die Belegung des Arrays mit den Positionen aller freien Zellen erfolgt bei jedem Aufruf der Funktion `findFreeCell`, die nach Berechnung der freien Zellen eine per Zufall auswählt. Nachdem ein Level fertig gespielt ist, wird immer die Funktion `cleanupBoard` aufgerufen. In diese fügen Sie nun bitte noch folgenden Code ein, der den durch `initializeBoard` allozierten Speicher für die Freiliste wieder freigibt.

```
// free allocated memory for free_list
free(free_list);
```

Der Code für die neue Funktion `findFreeCell` wir Ihnen hier ebenfalls vorgegeben.

```
enum ResCodes findFreeCell(struct board* aboard, struct pos* apos) {
int y,x,count;
enum ResCodes res_code;

// Collect all free cells
count = 0;
for (y = 0; y < aboard->last_row+1; y++) {
for (x = 0; x < aboard->last_col+1; x++) {
if (aboard->cells[y][x] == BC_FREE_CELL) {
free_list[count].x = x;
free_list[count].y = y;
count++;
}
}
}
if ( count == 0 ) {
// No more free cells
res_code = RES_NO_FREE_CELL_ERROR;
} else {
// Randomly select a cell from the free list
*apos = free_list[  rand() %  count ];
res_code = RES_OK;
}
return res_code;
}
```

Damit der Zufallsgenerator bei jedem Programmlauf eine andere Folge von Zufallszahlen erzeugt, müssen wir den Zufallsgenerator bei jedem Start unseres Programms mittels Funktion `srand` mit einem sich stets ändernden Wert initialisieren (einem sogenannten *seed* = Same). Fügen Sie folgende Zeile zu Beginn der Funktion `main` in `worm.c` ein:

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

Der Aufruf der Funktion `time(NULL)` liefert die Anzahl der seit dem 1.1.1970 vergangenen Sekunden.

Damit Sie die Funktion `time` benutzen können, müssen Sie in `worm.c` folgende `#include`-Direktive hinzufügen:

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

Zwischenbilanz

Falls Sie die bisher beschriebenen Änderungen vorgenommen haben und insbesondere die Funktionen des Moduls `sysworm` sinnvoll implementiert haben, sollte Ihr Projekt wieder fehlerfrei übersetzbar sein.
Wenn Sie Ihr Programm wie folgt aufrufen, sollten Sie 10 Systemwürmer in Aktion sehen:
`bin/worm -A10 -a`
Je nachdem, wie Sie Ihre Funktionen `growWorm` und `removeWorm` in der Vergangenheit implementiert haben, verhalten sich die Systemwürmer auch fehlerfrei. Diese Funktionen werden nun auch für die Systemwürmer benutzt. Falls bei Ihrer Implementierung manchmal Teile eines Wurms nicht korrekt vom Brett entfernt werden, müssen Sie noch kleine Korrekturen in den Funktionen `growWorm` und/oder `removeWorm` vornehmen. Nachfolgend finden Sie zusätzliche Erweiterungen, Korrekturen und Abrundungen des Codes.

Aufgabe 6 (Option -u: Benutzerwurm optional)

In dieser Aufgabe aktivieren wir die Option `-u`, die bereits im Modul `options` unterstützt wird. Falls der Schalter `-u` gewählt wird, soll der Benutzerwurm nicht sichtbar sein, d.h. es soll gar keinen geben. Diese Option wählen Sie zum Beispiel, wenn Sie ihr Wurm-Programm als Bildschirmschoner verwenden wollen.
Für diese Erweiterung muss lediglich die Funktion `doLevel` in der Datei `worm.c` angepasst werden. Die Änderungen am Code der Funktion `doLevel` werden im Folgenden angegeben:
Zunächst einmal behandeln wir den Benutzerwurm nun auch wie einen Systemwurm. Wir setzen die neuen Funktionen ein, die wir für die Systemwürmer geschrieben haben. Entfernen Sie folgenden Code in der Funktion `doLevel`:

```
// There is always an initialized user worm.
// Initialize the userworm with its size, position, heading.
// Assumption:
bottomLeft.y =  getLastRowOnBoard(&theboard);
bottomLeft.x =  0;
res_code = initializeWorm(&userworm,
    (theboard.last_row+1) * (theboard.last_col+1),
WORM_INITIAL_LENGTH, bottomLeft , WORM_RIGHT, COLP_USER_WORM);
if ( res_code != RES_OK) {
  return res_code;
}
// Show worm at its initial position
showWorm(&theboard, &userworm);
```

Und ersetzen Sie dafür:

```
// Initialize the userworm if it is active
if ( somegops->activate_userworm ) {
  res_code = initializeAndShowAutoWorm(
      &theboard,&userworm, WORM_INITIAL_LENGTH,COLP_USER_WORM);
  if ( res_code != RES_OK) {
    return res_code;
  }
}
```

Damit wird die Variable `struct pos bottomLeft` der Funktion `doLevel` überflüssig.
Entfernen Sie die Variable.

In der Hauptschleife der Funktion `doLevel` soll die Behandlung des Benutzerwurms optional sein, das heißt, sie soll davon abhängen, ob der Benutzerwurm sichtbar ist. Passen Sie den Code entsprechend folgender Vorlage an:

```
// Process userworm if it is active
if ( somegops->activate_userworm ) {
  // Clean the tail of the worm
  ...
  showWorm(&theboard, &userworm);
  // END process userworm
}
```

Analog müssen alle weiteren Anweisungen in der Funktion `doLevel`, die den Benutzerwurm betreffen, optional werden.

```
if ( somegops->activate_userworm ) {
  // Inform user about position and length of userworm . . .
  showStatus(&theboard, &userworm);
}
```

```
...
if ( somegops->activate_userworm ) {
  // Are we done with that level?
  if (getNumberOfFoodItems(&theboard) == 0) {
    end_level_loop = true;
  }
}
```

Die Aufräumarbeiten am Ende der Funktion `doLevel` dürfen nun nur noch ausgeführt werden, falls der Benutzerwurm aktiv ist. Ansonsten erhalten Sie eine Fehlermeldung bei der Freigabe des Speichers.

```
if ( somegops->activate_userworm ) {
  // Remove the worm from display and board
  if (userworm.wormpos != NULL) {
    removeWorm(&theboard, &userworm);
    // Because initializeWorm allocates memory
    cleanupWorm(&userworm);
  }
  clearMessageArea();
}
```

Der Aufruf der neuen Funktion `clearMessageArea` aus dem Modul `messages` hat nur kosmetische Gründe. Damit wird die Message-Area komplett gelöscht.

Aufgabe 7 (Benutzerwurm während des Spiels ein-/ausblenden)

Diese Aufgabe ist eng verwandt mit der letzten Aufgabe, bei der über die Option `-u` gesteuert wurde, ob der Benutzerwurm bei Spielbeginn erzeugt. In dieser Aufgabe fügen wir ein kleine Erweiterung hinzu, die es uns erlaubt, den Benutzerwurm während des Spiels durch Betätigung der Taste [u] ein- oder auszublenden.
Die Änderungen hierzu betreffen nur die Funktion `readUserInput` der Datei `worm.c`. Zur Umsetzung benötigen wir in der Funktion `readUserInput` Zugriff auf die Spieloptionen und auch Zugriff auf das Spielbrett, da wir den Wurm vom Spielbrett löschen oder ihn neu dort platzieren müssen.
Fügen Sie folgenden Code zur Funktion `readUserInput` hinzu und vergessen Sie nicht, den Aufruf der Funktion `readUserInput` entsprechend anzupassen:

```
void readUserInput(struct worm* aworm, enum GameStates* agame_state,
    struct game_options* somegops, struct board* aboard){
  ...
  case 'u' :    // User toggles activity of userworm
    if ( somegops->activate_userworm ) {
      // Disable userworm
      // Remove the worm from display and board
      if (aworm->wormpos != NULL) {
        removeWorm(aboard, aworm);
        // Because initializeWorm allocates memory
        cleanupWorm(aworm);
      }
      clearMessageArea();
      somegops->activate_userworm = false;
    } else {
      // Enable userworm
      enum ResCodes res_code =
      initializeAndShowAutoWorm(aboard,aworm,
      WORM_INITIAL_LENGTH, COLP_USER_WORM);
      if ( res_code != RES_OK) {
        *agame_state = WORM_USERWORM_ERROR;
      }
      somegops->activate_userworm = true;
    }
    break;
```

Aufgabe 8 (Systemwürmer während des Spiels ein/ausblenden)

Analog zum Benutzerwurm schaffen wir in dieser Aufgabe die Möglichkeit, alle Systemwürmer durch Betätigung der Taste (a) während des Spiels ein- oder auszublenden. Das geht aber nur, falls die Option `-A n` mit einer Zahl $n > 0$ beim Start des Spiels vereinbart wurde.
Auch diese Erweiterung kann durch eine einfache Änderung der Funktion `readUserInput` realisiert werden. Wir müssen nur die bereits vorhandenen Funktionen zu Manipulation der Systemwürmer einsetzen. Für die Erweiterung benötigen wir in der Funktion `readUserInput` Zugriff auf das Array der Systemwürmer, das heißt wir benötigen einen weiteren Parameter.
Fügen Sie folgenden Code zur Funktion `readUserInput` hinzu und vergessen Sie nicht, den Aufruf der Funktion `readUserInput` entsprechend anzupassen:

```
void readUserInput(struct worm* aworm, enum GameStates* agame_state,
    struct game_options* somegops, struct board* aboard,
    struct worm sysworms[] ) {
  ...
  case 'a' :    // User toggles activity of sysworms
    if ( somegops->activate_sysworms ) {
      // Disable sysworms
      disableSysworms(somegops,aboard,sysworms);
      somegops->activate_sysworms = false;
    } else {
      // Enable sysworms
      enum ResCodes res_code =
          enableSysworms(somegops,aboard,sysworms);
      if ( res_code != RES_OK) {
        *agame_state = WORM_SYSWORM_ERROR;
      }
      somegops->activate_sysworms = true;
    }
    break;
```

Bitte beachten Sie, dass durch den Aufruf der Funktion `disableSysworms` die Systemwürmer nicht nur unsichtbar werden, sondern dass sie vollständig zerstört werden. Analog dazu erzeugt der Aufruf von `enableSysworms` wieder ganz neue Systemwürmer in der gewünschten Zahl.

Aufgabe 9 (Cheat `-x`)

Da wir nun in der Funktion `readUserInput` Zugriff auf das Spielbrett haben, können wir sehr einfach folgenden Cheat einbauen. Bei Betätigung der Taste (x) wird so getan, als ob alle Futterbrocken bereits vertilgt wären. Das ist sehr praktisch zum Testen der Funktion `playGame`, die über alle Level iterieren soll.
Um den Cheat einzubauen, müssen Sie folgenden Code zur Funktion `readUserInput` hinzufügen:

```
case 'x' : // For development: force next level
  setNumberOfFoodItems(aboard,0);
  break;
```

Aufgabe 10 (Verbesserung von `growWorm` / Bugfix)

Eventuell bemerken Sie beim Spielen, dass bisweilen Teile von Würmern auf dem Spielfeld liegen bleiben. Etwa dann, wenn für einen Systemwurm keine gangbare neue Richtung gefunden werden kann. Dann wird der Wurm vom Brett genommen und neu an andere Stelle erzeugt.
Wenn beim Entfernen des Wurms vom Brett Teile des Wurms liegen bleiben, dann sind Ihre Funktionen `removeWorm` und `growWorm` nicht korrekt aufeinander abgestimmt. Falls `removeWorm` sich beim

Löschen vom Kopf des Wurms zum Schwanz vorarbeitet, müssen Sie garantieren, dass dabei zwischenzeitlich keine unbenutzten Positionen (`UNUSED_POS_ELEM`) passiert werden. In diesem Fall hört nämlich der normale Löschalgorithmus vorzeitig auf und es bleiben Wurmelemente liegen.
Eine solche Konstellation kann direkt nach dem Wachstum des Wurms entstehen (`growWorm`), wenn Sie nicht alle vor dem Kopf liegenden Elemente des Wurms im Ringpuffer ganz ans neue Ende des Puffers schieben. Also entweder die Funktion `removeWorm` robuster machen oder, was eleganter ist, die Funktion `growWorm` verbessern.
Die elegantere Lösung hat den zusätzlichen schönen Effekt, dass Ihr Wurm sofort sichtbar wächst, nachdem er einen Futterbrocken vertilgt hat.

Aufgabe 11 (Bugfix für `doLevel`)

Durch die oben beschriebenen Erweiterungen hat sich ein Fehler in der Logik der Funktion `doLevel` eingeschlichen. Bisher steht nach dem Aufruf der Funktion `readUserInput` folgende Überprüfung:

```
readUserInput(&userworm, agame_state, somegops, &theboard, sysworms);
if ( *agame_state == WORM_GAME_QUIT ) {
  end_level_loop = true;
  continue; // Go to beginning of the loop's block . . .
}
```

Das ist nicht mehr korrekt, da die Funktion `readUserInput` inzwischen auch andere Status-Codes in die Variable `*agame_state` schreibt. Es können jetzt dort auch die Fehler-Codes `WORM_SYSWORM_ERROR` und `WORM_USERWORM_ERROR` stehen, die durch die neuen Manipulationen der Systemwürmer oder des Benutzerwurms auftreten können.
Nehmen Sie folgende Korrektur vor:

```
readUserInput(&userworm, agame_state, somegops, &theboard, sysworms);
if (  *agame_state != WORM_GAME_ONGOING ) {
  end_level_loop = true;
  continue; // Go to beginning of the loop's block . . .
}
```

Aufgabe 12 (Abrundung)

Die Erweiterungen haben zur Folge, dass es ein paar neue Codes sowohl für die Aufzählung **enum** `GameSates` als auch für die Aufzählung **enum** `ResCodes` gibt. Zur Abrundung sollten wir an geeigneter Stelle auf diese neuen Codes adäquat reagieren. Das kann zum Beispiel so erfolgen: In der Funktion `doLevel` in der **switch**-Anweisung, die die Belegung der Variable `*agame_state` untersucht:

```
switch (*agame_state) {
  ...
  case WORM_SYSWORM_ERROR:
  showDialog("Es gibt ein Problem mit einem der Systemwuermer",
  "Bitte Taste druecken");
  res_code = RES_SYSWORM_ERROR;
  break;
  case WORM_USERWORM_ERROR:
  showDialog("Ihrem Wurm konnte leider keine Position zugewiesen
  werden", "Bitte Taste druecken");
  res_code = RES_USERWORM_ERROR;
  break;
```

In der Funktion `playGame`, nach Verlassen der Schleife über alle Level:

```
// Check why we left the loop
if ( res_code != RES_OK) {
  switch(res_code) {
    case RES_NO_FREE_CELL_ERROR:
      showDialog("Auf dem Board ist leider kein Platz mehr",
          "Bitte Taste druecken");
      break;
    case RES_NO_MEMORY:
      showDialog("Leider ist kein Arbeitsspeicher mehr
          verfuegbar", "Bitte Taste druecken");
      break;
    case RES_USERWORM_ERROR:
      showDialog("Es gibt ein Problem mit Ihrem Wurm",
          "Bitte Taste druecken");
      break;
    case RES_SYSWORM_ERROR:
      showDialog("Es gibt ein Problem mit einem der
          Systemwuermer", "Bitte Taste druecken");
      break;
    default:
      showDialog("Das Spiel wurde aufgrund eines Fehlers im
          Ablauf abgebrochen", "Bitte eine Taste druecken");
  }
} else if (level_list[cur_level] == NULL  &&
  game_state == WORM_GAME_ONGOING ) {
  ...
```

Eine letzte kosmetische Reparatur bietet sich auch für die Header-Datei `worm.h` an. Seit wir die Größe des Spielfeldes an die tatsächliche Größe des Anzeigefensters anpassen, gibt es eigentlich keinen Grund mehr für die Festlegung der Mindestanzahl von 26 Zeilen und 70 Spalten. Man könnte jede beliebige andere Mindestgröße festlegen, mit der sich vernünftig spielen lässt. Etwa:

```
#define MIN_NUMBER_OF_ROWS 10  // The guaranteed number of rows
#define MIN_NUMBER_OF_COLS 40  // The guaranteed number of columns
```

Aufgabe 13 (statisch binden)

Im Template `Worm100Template.zip` habe Sie auch ein erweitertes `Makefile` erhalten. Das neue `Makefile` erlaubt die Erzeugung einer statisch gebundenen ausführbaren Datei des Wurm-Programms. Das bedeutet, dass die benutzten Funktionen der C-Standard-Bibliothek und der Curses-Bibliotkek nicht dynamisch erst während der Laufzeit des Programms hinzu gebunden werden, sondern fest zur ausführbaren Datei gebunden werden (*statisch*).

Die statische Bindung von Programmen stellt das klassische Verfahren der Code-Bindung dar. Es hat aber den Nachteil, dass jedes noch so kleine ausführbare Programm alle aus Bibliotheken genutzte Funktionen fest eingebunden hat. Man hat dann in tausenden von Programmen immer den gleichen Code von z.B. der `printf`-Funktion gebunden. Das macht die ausführbaren Programme unnötig groß und verschwendet auch Arbeitsspeicher.
Das modernere Verfahren der dynamischen Bindung von Bibliotheken hat den Vorteil, dass der Code jeder benutzten Bibliotheksfunktion nur einmal in den Arbeitsspeicher geladen wird und dort von allen Programmen gemeinsam genutzt wird (*Code-Sharing, Shared Libraries*). Voraussetzung ist dann aber, dass auf dem System alle benötigten Bibliotheken als dynamisch bindbare Bibliotheken installiert sein müssen.
Dynamische Bindung spart also Platz, macht aber Voraussetzungen an das System, auf dem das Programm ausgeführt werden soll. Statische Bindung verschwendet mehr Platz, hat aber den Vorteil, dass das ausführbare Programm den kompletten aus Bibliotheken genutzten Code eingebunden hat.

Durch den Aufruf von `make dist` erzeugen Sie ein statisch gebundenes Programm und ein Unterverzeichnis `dist`, das neben dem Wurm-Programm auch alle Level-Dateien und die Dokumentation `usage.txt` enthält. Die so erzeugte Distribution läuft auf jedem Rechner, der ein zur ausführbaren Datei passendes Betriebssystem hat.

Der Befehl `make distclean` löscht das Unterverzeichnis `dist` mit seinem Inhalt.

Aufgabe 14 (Fehlerbehandlung, kontrollierte Freigabe von Ressourcen)

Wenn Sie den Code Ihrer Endversion in der Gesamtschau betrachten, werden Sie feststellen, dass die Prüfung der Resultate von Funktionsaufrufen und eine eventuell fällige Fehlerbehandlung einen nicht unwesentlichen Teil des Codes einnehmen.
An den Stellen, wo die Allokation von Speicher fehlschlägt, haben wir es uns sogar manchmal zu einfach gemacht und haben das Programm schlicht mittels `exit(CODE)` verlassen.
Ein derartiges Vorgehen ist für Desktop-Anwendungen vertretbar, die selten mehr als ein paar Stunden laufen. Für Server-Software ist dieser Weg nicht gangbar. Server-Prozesse laufen buchstäblich mehrere Jahre ohne Neustart. Daher müssen dort alle nicht mehr benötigten Ressourcen sorgfältig wieder an da Betriebssystem zurückgegeben werden. Neben Arbeitsspeicher sind Ressourcen zum Beispiel auch geöffnete Dateien oder Verbindungen zu Datenbanken. Das bedeutet aber, dass Sie bei der Allokation dreier Ressourcen `A`, `B`, `C` nach erfolgreicher Allokation von `A`, `B` und fehlgeschlagener Allokation von `C` nicht einfach per `exit` das Programm verlassen dürfen, sondern vorher die bereits allozierten Ressource `B` und `A` auch wieder freigeben müssen. In den meisten Fällen sogar in der umgekehrten Reihenfolge der Allokation.
Die damit einhergehende Fehlerbehandlung wird bei *naiver Programmierung* (so wie wir das gemacht haben) sehr schnell unübersichtlich. Im professionellen Umfeld setzt man stattdessen einen Ressourcen-Manager ein, der die bereits allozierten Ressourcen samt Reihenfolge verwaltet und bei Bedarf dann geordnet im Rahmen eines sogenannten `exit`-*Handlers* wieder freigibt.
Ein dementsprechendes einfaches Framework für die Programmierung in C finden interessierte Leser in dem äußerst empfehlenswerten Buch von Marc Rochkind über die fortgeschrittene Unix Programmierung. Im Umfeld der objektorientierten Programmierung gibt es spezielle Klassen für die Ressourcen-Verwaltung.

Literatur: *Mark J. Rochkind*: **Advanced UNIX Programming**, Second Edition, Addison-Wesley, 2008