Studiengänge IB, FFI, CSI
WS 2023/24

Technische Hochschule Ingolstadt
Fakultät für Informatik

# Praktikum zu „Grundlagen der Programmierung“

Blatt 8 (Vorführaufgabe)

**Lernziele:** Verwendung eines zweidimensionalen Arrays
**Erforderliche Kenntnisse:** Inhalte aus der Vorlesung zum Thema *mehrdimensionale Arrays*

**Voraussetzungen:**

- Vollständige Bearbeitung des letzten Blattes 07 (`Worm050`).

## Übersicht

Im Rahmen dieses Aufgabenblatts werden wir die Modellierung des Spielbretts verbessern. Wir platzieren Gegenstände auf dem Spielbrett, die zum einen Hindernisse darstellen, die der Wurm nicht berühren darf, zum anderen Futterbrocken sind, die der Wurm möglichst schnell aufsuchen und fressen soll. Abhängig vom Gehalt des Futterbrockens wächst der Wurm nach jeder Mahlzeit um eine bestimmte Anzahl von Gliedern.
Die technische Realisierung der Platzierung von Hindernissen und Futterbrocken auf dem Spielbrett basiert im Wesentlichen auf einer Datenstruktur, die für jede Position (Zeile, Spalte) auf dem Spielbrett speichert, durch welchen Gegenstand sie belegt ist. Der Gegenstand kann dabei ein Futterbrocken oder ein Teil eines Hindernisses oder aber ein Element des Wurms (später auch gegnerischer Würmer) sein.
In einer ersten einfachen Version (`Worm070`) werden wir die Größe des Spielfeldes fest vorgeben und auch die Gegenstände auf dem Spielbrett explizit im Programm kodieren. In einer späteren Version (`Worm090`) werden wir die Konfiguration des Spielfeldes aus beliebigen Dateien laden können. Dafür brauchen wir aber dann Datei-Operationen und dynamische Datenstrukturen – beides Konzepte, die wir bisher noch nicht in der Vorlesung behandelt haben.
Das folgende Bild zeigt das neue Spielbrett mit Hindernissen (`#`) und Futterbrocken (`2`, `4`, `6`) nebst Wurm (`'oooooo0`). Das Spielbrett hat in der Version Worm070 die feste Größe von 26 Zeilen und 70 Spalten, wobei die untere und die rechte Grenze durch Barrieren (`#`) optisch angezeigt werden. Die tatsächliche Größe des Fensters spielt daher keine Rolle, solange die Anzahl der Zeilen und Spalte grösser ist als die Mindestgröße von 30 Zeilen und 70 Spalten. Hierbei bildet die untere Barriere die erste Zeile der Message Area.

## Aufgabe 1 (Kopieren, Auspacken und Umbenennen des Templates)

Im Moodle-Kursraum finden Sie im Verzeichnis Praktikum die Datei `Worm070Template.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`:

```
$ cd ~/GdP1/Praktikum/Code
```

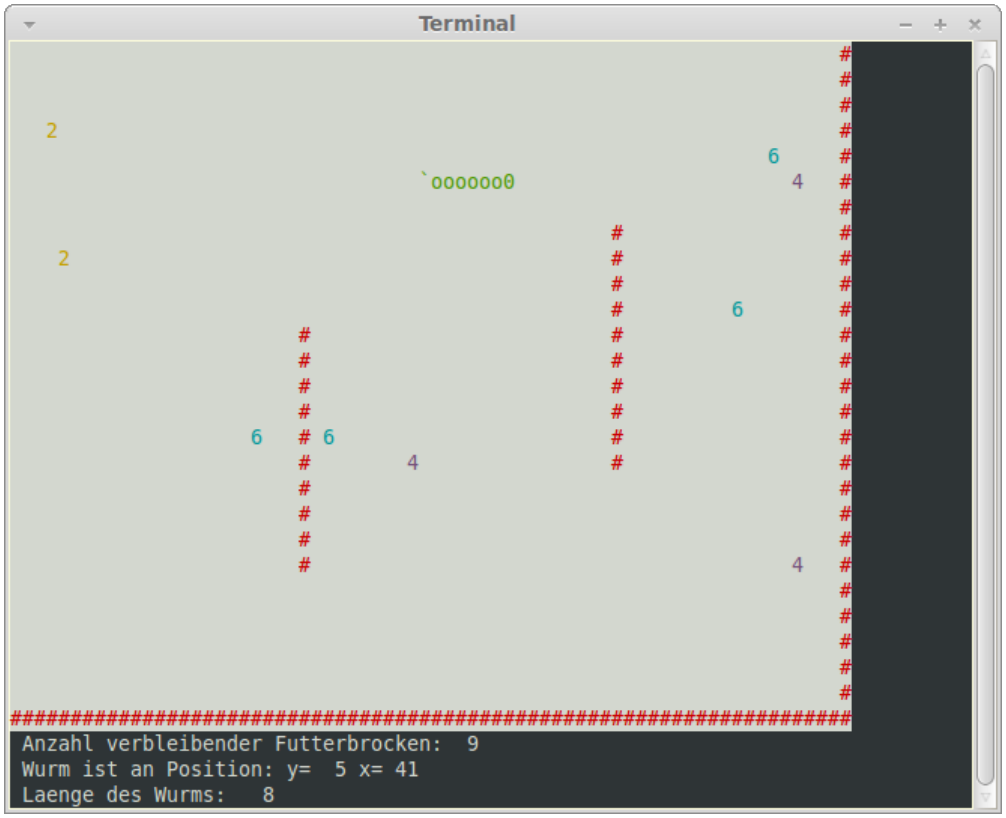


Abbildung 1: Das Spiel `worm` Nach Beendigung von Blatt 8.

Vergewissern Sie sich mittels des Befehls ls, dass die Zip-Datei im Verzeichnis vorhanden ist.
`$ ls Worm070Template.zip`
Entpacken Sie die Zip-Datei durch den nachfolgenden Befehl. Dadurch wird das Verzeichnis `Worm070Template` mit einigen Dateien darin angelegt.
`$ unzip Worm070Template.zip`
Listen Sie den Inhalt des neuen Unterverzeichnisses.
`$ ls -la Worm070Template`
Benennen Sie das Verzeichnis `Worm070Template` um in `Worm070`.
`$ mv Worm070Template Worm070`
Listen Sie den Inhalt des umbenannten Unterverzeichnisses.
`$ ls -la Worm070`
Listen Sie den Inhalt des aktuellen Verzeichnisses.
`$ ls -la`
Löschen Sie das Zip-Archiv
`$ rm -f Worm070Template.zip`
Führen Sie den nachfolgenden Kopierbefehl[1] aus. Dadurch werden die drei Dateien

- board_model.c
- worm_model.c
- worm.c

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

aus dem Verzeichnis `Worm050` in das neue Verzeichnis `Worm070` übernommen (kopiert).
Diese Dateien bilden die Ausgangssituation für die Neuerungen in der Version `Worm070`.

```
$ cp Worm050/board_model.c Worm050/worm*.c Worm070
```

## Aufgabe 2 (Test der Ausgangssituation)

Öffnen Sie eine Shell und wechseln Sie in dieser Shell ins Verzeichnis
`~/GdP1/Praktikum/Code/Worm070`.
Eine Auflistung der Dateien in diesem Verzeichnis sollte folgenden Inhalt anzeigen:

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

Bis auf die im letzten Schritt aus dem Verzeichnis `Worm050` kopierten drei Dateien (rot) sind alle anderen Dateien durch das Auspacken des Templates `Worm070Template.zip` entstanden.
In der nun hergestellten Ausgangssituation kann das Programm **nicht** fehlerfrei übersetzt werden, da die Dateien aus dem Template bereits Funktionalität voraussetzen, die Sie erst noch in den oben rot markierten Modul-Dateien implementieren müssen.

## Vorbetrachtung (Neuerungen in den Header-Dateien)

Um Ihre Implementierung für nachfolgende Aufgabenblätter zu fokussieren und den Arbeitsaufwand für dieses Arbeitsblatt im üblichen Rahmen zu halten, werden Ihnen die Header-Dateien der Version `Worm070` im Template vorgegeben. Kleinere Änderungen ihrerseits an diesen Vorgaben sind selbstverständlich nicht nur möglich, sondern werden explizit begrüßt. In den Aufgaben auf kommenden Arbeitsblättern gehen wir jedoch stets von den hier gemachten Vorgaben aus.
Das soll Sie aber trotzdem nicht vom Experimentieren abhalten!
Zunächst machen wir uns mit einigen der neuen Definitionen und Deklarationen in den Header-Dateien vertraut, damit Sie die Aufgaben dieses Blatts ausführen können. Die restlichen Änderungen besprechen wir dann jeweils im Kontext der einzelnen Teilaufgaben.
Bitte öffnen Sie nun die beiden Header-Dateien `board_model.h` und `worm.h` in Ihrem Editor.
Im Fokus dieses Aufgabenblatts steht die Einführung einer Datenstruktur **`struct board`** für das Spielbrett.
Diese Datenstruktur wird in der Header-Datei `board_model.h` des Moduls `board_model.c` definiert.

```
struct board {
        int last_row; // Last usable row on the board
        int last_col; // Last usable column on the board

        enum BoardCodes cells[MIN_NUMBER_OF_ROWS][MIN_NUMBER_OF_COLS];

        int food_items; // Number of food items left in the current level
};
```

Die Struktur enthält unter anderem ein zweidimensionales Array `cells`, welches für jedes Koordinatenpaar (`zeile`, `spalte`) einen Eintrag `cells[zeile][spalte]` enthalten soll. Dieser

Eintrag ist ein Element des Datentyps **enum** BoardCodes, der ebenfalls in board_model.h definiert ist:

```
enum BoardCodes {
        BC_FREE_CELL,     // Cell is free
        BC_USED_BY_WORM,  // Cell occupied by worm
        BC_FOOD_1,        // Food type 1; if hit by worm -> bonus of type 1
        BC_FOOD_2,        // Food type 2; if hit by worm -> bonus of type 2
        BC_FOOD_3,        // Food type 3; if hit by worm -> bonus of type 3
        BC_BARRIER        // A barrier; if hit by worm -> game over
};
```

Mit Hilfe des Datentyps **enum** BoardCodes speichern wir im Array cells, was sich auf dem Spielbrett in der Zelle mit den Koordinaten (zeile, spalte) befindet. Die Zelle ist entweder leer (BC_FREE_CELL), wird durch ein Element (Glied) eines Wurms belegt (BC_USED_BY_WORM), enthält einen Futterbrocken (BC_FOOD_1 -BC_FOOD_3) oder wird durch ein Hindernis belegt (BC_BARRIER).
Wie eingangs erwähnt, geben wir dem Spielbrett eine feste Größe von MIN_NUMBER_OF_ROWS Zeilen und MIN_NUMBER_OF_COLS Spalten, was direkt an der Definition des Arrays cells ersichtlich ist. Die CPP-Konstanten sind in der Datei worm.h definiert und legen, wie in früheren Versionen, die minimale Anzahl an Zeilen und Spalten fest, die das Anzeigefenster haben muss. Das Anzeigefenster muss mindestens so groß sein, dass das ganze Spielbrett und auch die Message-Area darin dargestellt werden können.
Um den Code unabhängig von den derzeit statisch vorgegebenen Grenzen des Arrays für Zeilen und Spalten (MIN_NUMBER_OF_ROWS und MIN_NUMBER_OF_COLS) zu halten, hat die Struktur **struct board** die beiden Komponenten last_row und last_column. Diese legen den jeweils größten Array-Index fest, den wir zum Zugriff auf die beiden Dimensionen des Arrays verwenden dürfen. Der Wert von last_row darf höchstens MIN_NUMBER_OF_ROWS - 1 sein und der Wert von last_column höchstens MIN_NUMBER_OF_COLS - 1. Später (Worm090) werden wir uns durch die Verwendung dynamisch allozierter Arrays ganz unabhängig von den derzeit statischen Array-Grenzen machen.
In der Komponente food_items speichern wir die Anzahl der sich noch auf dem Spielbrett befindlichen Futterbrocken.
Der Typ **enum** BoardCodes legt fest, **was** sich auf dem Spielbrett befinden kann. Wie die einzelnen Dinge **aussehen** sollen, definieren wir, wie bisher, durch die CPP-Konstanten mit dem Präfix SYMBOL_, die in worm.h definiert sind:

```
#define SYMBOL_FREE_CELL  ' '
#define SYMBOL_BARRIER   '#'
#define SYMBOL_FOOD_1    '2'
#define SYMBOL_FOOD_2    '4'
#define SYMBOL_FOOD_3    '6'
#define SYMBOL_WORM_HEAD_ELEMENT '0'
#define SYMBOL_WORM_INNER_ELEMENT 'o'
#define SYMBOL_WORM_TAIL_ELEMENT '`'
```

Die hier praktizierte Trennung zwischen Codes für die Belegung von Zellen des Spielbretts und der Kodierung der Darstellung durch Symbole mag auf den ersten Blick unnötig erscheinen. Es handelt sich hierbei aber um ein sehr wichtiges Design-Prinzip, das Sie in höheren Semestern, etwa in der Vorlesung *Software-Engineering*, unter dem Stichwort *MVC (Model-View-Control Pattern)* kennen lernen werden.
Im konkreten Fall erkennen wir die Nützlichkeit dieser Trennung, wenn wir den Code

BC_USED_BY_WORM und die Konstanten SYMBOL_WORM_HEAD_ELEMENT, SYMBOL_WORM_INNER_ELEMENT und SYMBOL_WORM_TAIL_ELEMENT betrachten. Über den Code speichern wir im Array cells, dass die Zelle durch ein Wurm-Element belegt ist. Dabei ist egal, um welches Element des Wurms es sich handelt. Insbesondere muss der Code im Array so lange nicht geändert werden, solange irgendein Teil des Wurms diese Zelle belegt. Auf der anderen Seite können wir aber bei der Darstellung des Wurms für jedes einzelne Glied ein anderes Symbol wählen. Wir werden zum Beispiel Kopf und Schwanz des Wurms mit anderen Symbolen als die inneren Elemente des Wurms darstellen (siehe Aufgabe 9).
Die restlichen Neuerungen in den Header-Dateien werden im Rahmen der einzelnen Teilaufgaben besprochen.

# Aufgabe 3 (Setter und Getter in der Datei `board_model.c`)

Öffnen Sie bitte die Datei board_model.c, die wir aus Worm050 kopiert haben, in Ihrem Editor. In dieser Datei werden wir im Rahmen dieses Aufgabenblatts die meisten Änderungen vornehmen. Als Aufwärmübung implementieren wir zunächst ein paar Zugriffsfunktionen[2] (Setter/Getter), die in anderen Modulen genutzt werden, um auf die Komponenten der Datenstruktur **struct board** schreibend (Setter) und lesend (Getter) zuzugreifen. Die zugehörigen Funktionsdeklarationen finden Sie in board_model.h.
Als Getter (Abfragemethoden) werden im engeren Sinne alle Funktionen bezeichnet, die lediglich den Wert einer Komponente eines Verbunds zurückgeben. In einem weniger strengen Begriffsrahmen, der aber dem Kapselungsprinzip (Verstecken von Implementierungsdetails) besser Rechnung trägt, kann der zurückgegebene Werte auch aus einer Kombination von Komponentenwerten berechnet werden oder im Extremfall sogar unabhängig von Komponentenwerten sein.
Im Gegensatz dazu werden, wieder im engeren Sinn, Funktionen, die als Werte übergebene Parameter den Komponenten eines Verbunds zuweisen als Setter (Änderungsmethoden) bezeichnet. Im weniger strengen Begriffsrahmen werden Funktionen, die irgendwelche Änderungen an Komponenten vornehmen, als Setter bezeichnet. Dabei muss nicht notwendigerweise ein als Parameter übergebener Wert eine Rolle spielen.
Am häufigsten werden die Begriffe Setter und Getter im Rahmen der objektorientierten Programmierung verwendet. Statt von den Komponenten einer Struktur spricht man hier von den Attributen einer Klasse.
Die Bezeichnung Getter und Setter rührt daher, dass die Namen von Getter-Funktionen oft das Präfix get und die Namen von Setter-Funktionen oft das Präfix set haben.

## Funktion `getLastRowOnBoard`:

Löschen Sie die Implementierung der Funktion getLastRow und fügen Sie stattdessen folgenden Code ein:

```
int getLastRowOnBoard(struct board* aboard) {
        return aboard->last_row;
}
```

Die neue Funktion leistet im Prinzip das gleiche wie die alte Funktion. Zum einen hat sich aber

[2] Siehe auch https://de.wikipedia.org/wiki/Zugriffsfunktion.

der Name der Funktion geändert, zum anderen berechnen wir nicht mehr den Index der letzten erlaubten Zeile, sondern geben lediglich den Wert der Komponente `aboard->last_row` zurück. Der Wert dieser Komponente wird in der Funktion `initializeBoard` gesetzt, die Sie später implementieren werden.
Der Funktionsparameter `aboard` enthält die Adresse einer Strukturvariablen vom Typ **`struct board`**. Durch die Übergabe des Parameters `aboard` vermeiden wir die Verwendung einer globalen Variablen.
Vergleichen Sie hierzu bitte die Einführung des Funktionsparameters **`struct worm`**`* aworm` in den Funktionen, die Sie im Rahmen des Aufgabenblatts 7 implementiert haben. Es handelt sich um die gleiche Technik.

**Hinweis:** Wenn im Folgenden bei der Besprechung der einzelnen Funktionen von Komponenten die Rede ist, sollen immer die Komponenten der Struktur gemeint sein, deren Adresse im Parameter `aboard` übergeben wird.

## Funktion `getLastColOnBoard`:

Löschen Sie nun die Funktion `getLastCol` und implementieren Sie analog zu oben die neue Funktion

```
int getLastColOnBoard(struct board* aboard)
```

Die Funktion soll den Wert der Komponente `last_col` zurückgeben.

## Funktion `getNumberOfFoodItems`:

Implementieren Sie analog zu oben die neue Funktion

```
int getNumberOfFoodItems(struct board* aboard)
```

Die Funktion soll den Wert der Komponente `food_items` zurückgeben.

## Funktion `getContentAt`:

Nun fehlt noch eine lesende Zugriffsfunktion (Getter), dann haben wir alle Komponenten des Verbunds **`struct board`** durch lesende Zugriffsfunktionen abgedeckt.
Implementieren Sie die Funktion

```
enum BoardCodes getContentAt(struct board* aboard, struct pos position)
```

Die Funktion soll den Code zurückgeben, der im Array `cells` unter Zeile `position.y` und Spalte `position.x` gespeichert ist.
Achten Sie speziell auf die korrekte Syntax beim Zugriff auf das zweidimensionale Array `cells`!

### Funktion `setNumberOfFoodItems`:

Nun wenden wir uns den schreibenden Zugriffsfunktionen zu. Zunächst implementieren wir eine Funktion, die der Komponente `food_items` einen als Parameter übergebenen Wert zuweist (Setter). Implementieren Sie die Funktion

```
void setNumberOfFoodItems(struct board* aboard, int n)
```

Die Funktion soll den im Parameter `n` übergebenen Wert der Komponente `food_items` zuweisen.

### Funktion `decrementNumberOfFoodItems`:

Als nächstes implementieren wir eine Funktion, die die in der Komponente `food_items` gespeicherte Anzahl der noch auf dem Spielbrett verbleibenden Futterbrocken um eins erniedrigt (dekrementiert).
Implementieren Sie die Funktion

```
void decrementNumberOfFoodItems(struct board* aboard)
```

Die Funktion soll den Wert der Komponente `food_items` um 1 erniedrigen.

## Aufgabe 4 (Die Funktion `initializeBoard`)

Die Funktion `initializeBoard` wird aus der Hauptschleife des Spiels (`doLevel`) aufgerufen werden, um die dort definierte Strukturvariable `theboard` zu initialisieren. Die diesbezügliche Änderung an der Funktion `doLevel` erledigen wir in der letzten Teilaufgabe dieses Aufgabenblatts.
In der aktuellen Teilaufgabe werden wir die Funktion `initializeBoard` implementieren. Da wir das Array `cells` in der aktuellen Version mit den expliziten Grenzen `MIN_NUMBER_OF_ROWS` und `MIN_NUMBER_OF_COLS` definieren, prüfen wir zu Beginn der Funktion nochmals diese Grenzen. Falls die aktuelle Größe des Fensters nicht ausreichend ist, zeigen wir eine Fehlermeldung in der Message Area an und verlassen danach die Funktion mit einem Fehlercode. Ansonsten berechnen wir den jeweils größten Zugriffsindex für Zeilen und Spalten und weisen diese Werte den dafür vorgesehenen Komponenten zu.
Die Funktion `initializeBoard` wird erst bei der Einführung von dynamischen Arrays (`Worm090`) richtig interessant. Dann müssen wir nämlich in der Funktion ausreichend Speicher allozieren. Derzeit ist sie aber eher langweilig und die Aufgabenbeschreibung würde länger als die Implementierung dauern. Daher ist der Code der Funktion `initializeBoard` hier bereits vollständig abgedruckt. Sie müssen den Code nur noch in die Datei `board_model.c` einfügen.
Im Rahmen der Bearbeitung der Version `Worm090` dürfen Sie aber ganz bestimmt die Änderungen an der Funktion `initializeBoard` durchführen ;-)

```
enum ResCodes initializeBoard(struct board* aboard) {
    // Check dimensions of the board
    if ( COLS < MIN_NUMBER_OF_COLS  ||
    LINES < MIN_NUMBER_OF_ROWS  + ROWS_RESERVED ) {
        char buf[100];
        sprintf(buf,"Das Fenster ist zu klein: wir brauchen %dx%d",
        MIN_NUMBER_OF_COLS , MIN_NUMBER_OF_ROWS + ROWS_RESERVED );
        showDialog(buf,"Bitte eine Taste druecken");
        return RES_FAILED;
```

```
    }
    // Maximal index of a row
    aboard->last_row = MIN_NUMBER_OF_ROWS  - 1;
    // Maximal index of a column
    aboard->last_col = MIN_NUMBER_OF_COLS - 1;
    return RES_OK;
}
```

In der oben abgedruckten Funktion `initializeBoard` wird die Hilfsfunktion `showDialog` benutzt, welche im Modul `messages.c` definiert ist. Damit wir die Funktion aufrufen können, müssen wir vorher die Header-Datei des Moduls inkludieren. Fügen Sie am Anfang der Datei `board_model.c` noch eine geeignete *##include*-Direktive ein.

# Aufgabe 5 (Änderung in Funktion `placeItem`)

Die Funktion `placeItem` hatte bisher lediglich ein Symbol in einer bestimmten Farbe an einer (`y`, `x`) Position in den Fensterpuffer der Curses-Bibliothek geschrieben. Nun wird diese Funktion dahingehend erweitert, dass in der Datenstruktur für das Spielbrett vermerkt wird, dass die Position (`y`, `x`) durch einen Gegenstand belegt wird.
Besagte Datenstruktur für das Spielbrett ist eine Variable vom Typ **`struct board`**, die in der Hauptschleife des Spiels (`doLevel`) definiert wird und deren Adresse als Parameter `aboard` an die Funktion `placeItem` übergeben wird.
Studieren Sie in der Header-Datei `board\_model.h` die geänderte Deklaration der Funktion `placeItem`.
Die Funktion bekommt nun zwei neue Parameter

```
struct board* aboard
enum BoardCodes board_code
```

Passen Sie sodann die Kopfzeile der Funktion `placeItem` in der Modul-Datei `board_model.c` entsprechend an. Im Rumpf der Funktion müssen Sie dann noch eine neue Anweisung einfügen, die den Wert der Variable `board_code` in der Komponente `cells` an geeigneter Position abspeichert. Es soll gespeichert werden, dass an Position (`y`, `x`) auf dem Spielbrett nun ein Gegenstand liegt, der den Code `board_code` hat.
Folgende Zeile soll Ihnen eine Hilfestellung geben:
`aboard-> @000 = board_code;` Sie müssen das `@000` durch einen geeigneten Zugriff auf das zweidimensionale Array `cells` ersetzen.

# Aufgabe 6 (Die Funktion `initializeLevel`)

Nun wenden wir uns der zentralen Funktion des Moduls `board_model.c` zu.
Die Funktion `initializeLevel` hat die Aufgabe, das Spielbrett mit Gegenständen (Hindernissen und Futterbrocken) zu füllen. In einer ersten einfachen Ausführung werden wir die Platzierung der einzelnen Gegenstände explizit im Rumpf der Funktion `initializeLevel` implementieren. In einer späteren Version (`Worm090`) werden wir die Beschreibung eines Spiel-Levels aus einer Datei lesen können. Das erlaubt uns dann, verschiedene Spiel-Szenarien mit steigendem Schwierigkeitsgrad (Level) in Dateien zu hinterlegen und dann während des Spiels bei Bedarf nachzuladen.
Der Code der Funktion `initializeLevel` ist etwas umfangreicher als bei den anderen Funktionen, und daher bekommen Sie ein Template als Vorlage, das Sie an ein paar interessanten Stellen

noch ergänzen müssen. Die Stellen sind wie üblich durch die Platzhalter nnn gekennzeichnet (siehe auch Folgeseite).

```
enum ResCodes initializeLevel(struct board* aboard) {
    @001    // define local variables for loops etc
    // Fill board and screen buffer with empty cells.
    for (y = 0; @002 ; y++) {
        for (x = 0; @002 ; x++) {
            placeItem(aboard,y,x,BC_FREE_CELL,SYMBOL_FREE_CELL,COLP_FREE_CELL);
        }
    }
    // Draw a line in order to separate the message area
    // Note: we cannot use function placeItem() since the message area
    // is outside the board!
    y = aboard->last_row + 1;
    for (x=0;  @003 ; x++) {
        move(y, x);
        attron(COLOR_PAIR(COLP_BARRIER));
        addch(SYMBOL_BARRIER);
        attroff(COLOR_PAIR(COLP_BARRIER));
    }
    // Draw a line to signal the rightmost column of the board.
    for (y=0; y <= aboard->last_row ; y++) {
        placeItem(aboard,y,aboard->last_col,
        BC_BARRIER,SYMBOL_BARRIER,COLP_BARRIER);
    }
    // Barriers: use a loop
    for ( @004 ) {
        @004
        placeItem(aboard,y,x,BC_BARRIER,SYMBOL_BARRIER,COLP_BARRIER);
    }
    for ( @005 ) {
        @005
        placeItem(aboard,y,x,BC_BARRIER,SYMBOL_BARRIER,COLP_BARRIER);
    }
    // Food
    placeItem(aboard, 3, 3,BC_FOOD_1,SYMBOL_FOOD_1,COLP_FOOD_1);
    @006

    // Initialize number of food items
    // Attention: must match number of items placed on the board above
    aboard->food_items = 10;
    return RES_OK;
}
```

Bemerkungen zu den einzelnen Platzhaltern:

**@001:** In der Funktion `initializeLevel` werden Sie mehrere Schleifen programmieren. An dieser Stelle können Sie bei Bedarf die Schleifenvariablen deklarieren.

**@002:** In dieser zweifach geschachtelten Schleife sollen Sie jede (`y`,`x`) Position des Boards als leer markieren. Der Eintrag sowohl in der Strukturvariable des Spielbretts (`aboard`) als auch im Bildschirmpuffer wird bereits durch die Funktion `placeItem` erledigt. Man muss diese Hilfsfunktion lediglich mit den richtigen Parametern aufrufen. Sie müssen dafür sorgen, dass die Laufvariablen `y` und `x` alle erforderlichen Werte einnehmen.
Beachten Sie die Array-Grenzen und nutzen Sie die eigens hierfür angelegten Komponentenvariablen der Struktur **`struct board`**.

**@003:** Hier sollen Sie eine Schleife programmieren, die am unteren Ende des Spielbretts eine horizontale Linie bestehend aus Hindernissen zeichnet. Der Zeilenindex `lastrow + 1` bezeichnet hierbei gerade die erste Zeile der Message Area. Bitte beachten Sie, dass diese Zeile nicht mehr zum Spielbrett gehört! Wir wählen lediglich aus optischen Gründen das Symbol für Hindernisse, um die Trennlinie zwischen Spielbrett und Message Area zu kennzeichnen (siehe Screenshot Seite 1). Die Funktion `showBorderLine` des Moduls `messages` wir dadurch überflüssig und kann nebst Aufruf in Funktion `doLevel` gelöscht werden.
Weil dieser Zeilenindex nicht mehr zum Spielbrett gehört, können wir die Funktion `placeItem` nicht verwenden, da diese unter diesem Zeilenindex auf das Array `cells` zugreifen würde, was eine Überschreitung der Array-Grenzen zur Folge hätte. Der Code ist jedoch ähnlich zu dem der Funktion `placeItem`.
Danach finden Sie bereits vollständigen Code, der eine vertikale Linie in der letzten Spalte des Spielbretts `lastcol` zeichnet. Dieser Begrenzer ist notwendig, damit der Wurm nicht über den rechten Rand des Spielbretts entwischt. Bitte beachten Sie, dass das Anzeigefenster in der Version `Worm070` breiter sein kann als das Spielbrett (siehe Screenshot Seite 1).
Diese unbefriedigende Situation resultiert daher, dass wir derzeit nur Arrays mit statischen Grenzen zur Verfügung haben und uns daher nicht dynamisch auf die tatsächliche Größe des Anzeigefensters (`LINES` x `COLS`) einstellen können.

**@004:** Implementieren Sie hier ein Schleife, die eine senkrechte Barriere mit 10 bis 15 Zeichen erzeugt. Positionieren Sie die Barriere im linken Drittel des Spielbretts (siehe Screenshot auf Seite 1).

**@005:** Implementieren Sie hier ein Schleife, die eine senkrechte Barriere mit 10 bis 15 Zeichen erzeugt. Positionieren Sie die Barriere im rechten Drittel des Spielbretts (siehe Screenshot auf Seite 1).

**@006:** Hier sollen Sie insgesamt 10 Futterbrocken unterschiedlichen Typs auf dem Spielbrett positionieren. Das Template zeigt den Aufruf der Funktion `placeItem`, der einen Futterbrocken der Kategorie 1 an Position `(3, 3)` platziert. Die Kategorie wird durch die Verwendung der Konstanten `BC_FOOD_i`, `SYMBOL_FOOD_i` und `COLP_FOOD_i` gewählt, wobei `i` die Zahlen 1,2,3 einnehmen kann. Platzieren Sie nun durch analoge Aufrufe 9 weitere Futterbrocken nach folgender Aufstellung:

- 2 Futterbrocken der Kategorie 1 (einer wird schon durch das Beispiel platziert)
- 4 Futterbrocken der Kategorie 2
- 4 Futterbrocken der Kategorie 3

Die konkreten Positionen spielen keine Rolle, solange Sie immer Koordinaten wählen, die noch unbelegt sind und zum Spielbrett gehören. Sie können auch mehr als 10 Futterbrocken austeilen, müssen dann aber die Zuweisung der Anzahl der Futterbrocken an `food_items` entsprechend anpassen.
Als Fleißaufgabe können Sie die Positionen für die Futterbrocken auch zufällig wählen. Fragen Sie Ihren Betreuer, welche Bibliothek Ihnen einen Zufallsgenerator zur Verfügung stellt oder schlagen Sie die fehlende Information in der Beschreibung der Standardbibliothek nach (Tip: Funktion `rand`).

Hiermit sind die Änderungen im Modul `board_model.c` abgeschlossen. Wenden wir uns nun dem Modul `worm_model.c` zu. Neben Änderungen, die aus der Einführung der Struktur **`struct board`** resultieren, werden wir auch Funktionalität hinzufügen, die das Wachsen den Wurms als Folge der Vertilgung von Futterbrocken realisiert.

## Vorbetrachtungen zu Änderungen im Modul `worm_model.c`

Beim Studium der vorgegebenen Header-Datei `worm_model.h` sehen wir, dass einige Funktionen des Moduls `worm_model.c` einen neuen Parameter **`struct board`**`*` `aboard` erhalten haben. Dieser Parameter enthält immer die Adresse der Stukturvariable `theboard`, die in der Funktion `doLevel` definiert wird (siehe Teilaufgabe 15) und dort beim Aufruf der Funktionen des Moduls `worm_model.c` als Parameter mitgegeben wird.
Die Header-Datei enthält aber noch mehr Änderungen. Ein paar Deklarationen und Definition wurden aus der Header-Datei worm.h in die Header-Datei `worm_model.h` verschoben, weil sie dort thematisch besser aufgehoben sind. Aus dem gleichen Grund wurden andere Definitionen und Deklaration in die Header-Datei `board_model.h` verschoben.
Im Zuge der Version `Worm070` wollen wir den Effekt einbauen, dass der Wurm wachsen kann, wenn er Futterbrocken frisst. Die Animation des Wachstums lässt sich mit Hilfe unserer Datenstruktur **`struct worm`** für den Wurm sehr leicht realisieren.

Bisher haben wird den Wurm gleich von Beginn an den Ringpuffer `wormpos` vollständig füllen lassen. Wenn der Kopf des Wurms zu Beginn erscheint, ist nur das erste Element des Ringpuffers mit den Positionskoordinaten des Kopfs belegt. Die restlichen Elemente sind durch die Konstante `UNUSED_POS_ELEM` markiert. Dann belegt der Wurm mit jedem Schritt ein weiteres Element des Ringpuffers, bis der letzte erlaubte Zugriffsindex `maxindex` erreicht ist. Der Wurm ist damit in voller Länge sichtbar, und im nächsten Schritt wird, nach dem das Schwanzelement gelöscht wurde, der Ringpuffer wieder von vorne beschrieben.
Die Animation eines Wachstums implementieren wir, indem wir eine neue Grenze `cur_lastindex` als Komponente der Struktur **`struct worm`** einführen (siehe Header-Datei `worm_model.h`). In der Funktion `initializeWorm` setzen wir die neue Grenze `cur_last_index` (für *current last index*) auf einen kleineren Wert als `maxindex`. Statt wie bisher bei `maxindex` zwingen wir nun den Wurm schon bei Erreichen der kleineren Grenze `cur_lastindex` dazu, wieder am Anfang des Ringpuffers zu beginnen. Wir erlauben also dem Wurm nicht, den ganzen Ringpuffer zu benutzen. Dafür müssen wir nur im Code der Funktionen `cleanWormTail` und `moveWorm` die Grenze `maxindex` durch `cur_lastindex` ersetzen.
Wenn der Wurm aber einen Futterbrocken frisst, das heißt wenn der Kopf auf dem Spielbrett mit einem Futterbrocken kollidiert, dann erhöhen wir einfach die neue Grenze `cur_lastindex` um einen bestimmten Betrag, der von der Wertigkeit des Futterbrockens (Kategorie 1,2 oder 3) abhängt. Da der Wurm nun mehr Platz im Ringpuffer erhält, können mehr Positionen gespeichert werden, was nach ein paar Schritten dazu führt, dass der Wurm länger wird. Das maximale Wachstum des Wurms ist erreicht, wenn die Grenze `cur_lastindex` gleich groß wie die Grenze `maxindex` wird.
In den folgenden Teilaufgaben werden wir die hier skizzierten kleinen Änderungen vornehmen.

# Aufgabe 7 (Änderungen an der Funktion `initializeWorm`)

Die Funktion `initializeWorm` erhält einen neuen Parameter `int len_cur`, der die initiale Länge des Wurms angibt (siehe Deklaration in `worm_model.h`).
Passen Sie Ihre Implementierung in `worm_model.c` dementsprechend an.
Fügen Sie sodann im Rumpf der Funktion nach der Zeile

```
aworm->maxindex = len_max - 1 ;
```

folgende weitere Anweisung ein, die den letzten erlaubten Zugriffsindex im künstlich verkleinerten Ringpuffer in der neuen Komponente `cur_lastindex` speichert.

```
// Current last usable index in array. May grow upto maxindex
aworm->cur_lastindex = len_cur - 1;
```

Darüber hinaus sind keine weiteren Änderungen an der Funktion `initializeWorm` nötig

# Aufgabe 8 (Änderungen in der Funktion `cleanWormTail`)

In der Funktion `cleanWormTail` fallen nur drei kleine Änderungen an.

1. Die Funktion bekommt den zusätzlichen Parameter `struct board* aboard`.
   Passen Sie Ihre Implementierung in `worm_model.c` entsprechend der Vorgabe in der Header-Datei `worm_model.h` an.

2. Im Rumpf der Funktion wird `placeItem` aufgerufen. Diese Funktion hat aufgrund der Änderungen in `board_model.c` zwei neue Parameter (siehe `board_model.h`). Ändern Sie den Aufruf der Funktion so, dass der Parameter `aboard` und der Code `BC_FREE_CELL` an der richtigen Stelle im Aufruf der Funktion `placeItem` verwendet werden.

3. Die Logik der Funktion `cleanWormTail` nutzt bisher die Grenze `maxindex` für die Begrenzung des Ringpuffers. Ersetzen Sie diese Grenze durch die neue kleinere Grenze `cur_lastindex`, entsprechend den Vorüberlegungen.

# Aufgabe 9 (Änderungen in der Funktion `showWorm`)

In der Funktion `showWorm` ergeben sich durch die Einführung der Struktur `struct board` Änderungen in der Schnittstelle. Die Funktion hat nun den zusätzlichen Parameter `struct board* aboard` (siehe Header-Datei `worm_model.h`). Im folgenden ist der angepasste Code abgedruckt:

```
void showWorm(struct board* aboard, struct worm* aworm) {
    // Due to our encoding we just need to show the head element
    // All other elements are already displayed
    placeItem(aboard,
    aworm->wormpos[aworm->headindex].y,
    aworm->wormpos[aworm->headindex].x,
    BC_USED_BY_WORM,SYMBOL_WORM_INNER_ELEMENT,aworm->wcolor);
}
```

Bisher haben wir in der Funktion `showWorm` immer nur das neue Kopfelement ausgegeben, da die restlichen Elemente nicht neu gezeichnet werden müssen, weil alle Elemente gleich aussehen. Die restlichen sichtbaren Elemente des Wurms sind immer Darstellungen alter Kopfelemente. Nun wollen wir eine interessantere Darstellung des Wurms implementieren. Das Kopfelement soll durch das Symbol ‚**O**', das Schwanzelement durch das Symbol ‚**`**'und allen anderen Zwischenelemente durch das Symbol ‚**o**'dargestellt werden. Zu diesem Zweck werden in der Header-Datei `worm.h` folgende CPP-Konstanten definiert[3]:

```
#define SYMBOL_WORM_HEAD_ELEMENT 'O'
#define SYMBOL_WORM_INNER_ELEMENT 'o'
#define SYMBOL_WORM_TAIL_ELEMENT '`'
```

Erweitern Sie nun die oben abgedruckte Version der Funktion `showWorm` dahingehend, dass nicht nur das neue Kopfelement des Wurms mit `placeItem` ausgegeben wird, sondern diese Funktion für jedes Element des Wurms aufgerufen wird. Dabei soll dann, abhängig vom gerade benutzten Index für den Zugriff auf den Ringpuffer, das passende der drei obigen Symbole zur Ausgabe des Wurmelements verwendet werden.
Bitte beachten Sie, dass Sie unabhängig vom verwendeten Symbol für die Darstellung des Wurmelements immer den gleichen Code `BC_USED_BY_WORM` verwenden müssen.
Orientieren Sie sich bei Ihrer Lösung am Aufbau der Funktion `isInUseByWorm`. In dieser Funktion haben Sie bereits eine Schleife über alle Indexpositionen des Ringpuffers programmiert, um Kollisionen zu entdecken. Dort haben Sie wahrscheinlich beim Index des Kopfelements begonnen und haben dann den Ringpuffer rückwärts laufend durchsucht, bis Sie am Index des Schwanzelements angelangt waren (do-while-Schleife mit Dekrement der Indexposition nach jedem Schleifendurchlauf).
Denken Sie daran, dass Sie beim Durchlauf durch den Ringpuffer die Grenze `cur_lastindex` anstatt der Grenze `maxindex` verwenden müssen.

## Aufgabe 10 (Die Funktion `getWormLength`)

Diese Getter-Funktion soll es anderen Modulen erlauben, die aktuelle Länge eines Wurmes zu erfragen. Die Deklaration der Funktion in der Header-Datei `worm_model.c` legt folgendes Gerüst für die Funktion nahe:

```
int getWormLength(struct worm* aworm) {
        return @001
}
```

Füllen Sie den fehlenden Code anstelle des Platzhalters `@001` ein. Zur Berechnung der aktuellen Länge des Wurms sollten Sie die Komponente `cur_lastindex` heranziehen.

## Aufgabe 11 (Die Funktion `growWorm`)

Die Futterbrocken werden ausgelegt, damit der Wurm sich ernähren und wachsen kann. Für jeden gefressenen Futterbrocken gibt es einen Bonus. Zur Kodierung des Bonus-Systems wurde eigens ein neuer Aufzählungstyp in der Header-Datei `worm_model.h` definiert:

[3]Sie dürfen gerne beliebige andere Symbole verwenden. Sie können mit etwas Aufwand auch *Emojis* und weiter UTF-8 Characters verwenden.

```
enum Boni {
        BONUS_1 = 2, // additional length for worm when consuming food of type 1
        BONUS_2 = 4, // additional length for worm when consuming food of type 2
        BONUS_3 = 6, // additional length for worm when consuming food of type 3
};
```

Die Funktion `growWorm` implementiert das Wachsen des Wurms. Durch den Parameter `growth` vom Typ **`enum`** `Boni` ist festgelegt, um wie viele Glieder der Wurm wachsen soll. Wir werden die Funktion `growWorm` in der Funktion `moveWorm` aufrufen, wo wir abhängig vom gerade gefressenen Futterbrocken ein entsprechendes Argument vom Typ **`enum`** `Boni` wählen.
Um Sie für die nächste Teilaufgabe zu schonen, ist im Folgenden bereits der vollständige Code der Funktion `growWorm` abgedruckt. Fügen Sie den Code in Ihrer Datei `worm_mode.c` hinzu.

```
void growWorm(struct worm* aworm, enum Boni growth) {
        // Play it safe and inhibit surpassing the bound
        if (aworm->cur_lastindex + growth <= aworm->maxindex) {
                aworm->cur_lastindex += growth;
        } else {
                aworm->cur_lastindex = aworm->maxindex;
        }
}
```

# Aufgabe 12 (Änderungen in der Funktion **moveWorm**)

Nun wenden wir uns der letzten Funktion zu, die wir in der Modul-Datei `worm_model.c` ändern müssen.
In der Funktion `moveWorm` sind im Wesentlichen zwei Änderungen zu machen.
Erstens müssen wir hier die Verwendung der neuen Datenstruktur **`struct board`** für das Spielbrett einbauen, denn in `moveWorm` wird der Wurm um einen Schritt bewegt, wobei sowohl Futterbrocken als auch Hindernisse gerammt werden können.
Wir werden die bisher von der Funktion `isInUseByWorm` durchgeführte Prüfung auf Selbstkollision durch Logik in der Funktion `moveWorm` ersetzen, die den neuen Funktionsparameter **`struct board`**`* aboard` ausnützt. Diese neue Logik behandelt alle Fälle, in denen der Wurm mit anderen Gegenständen auf dem Spielbrett kollidiert.
Zweitens müssen wir, falls der Wurm mit einem Futterbrocken kollidiert, das entsprechende Wachstum durch Aufruf der Funktion `growWorm` anstoßen.
Die neue Kollisionsbehandlung in der Funktion `moveWorm` besteht im Wesentlichen aus einer großen Fallunterscheidung (**`switch`**). Abhängig davon, welcher Gegenstand sich an der neuen, vom `moveWorm` berechneten, Kopfposition befindet, werden unterschiedliche Aktionen ausgeführt.
Im Folgenden ist ein Template der Funktion `moveWorm` in mehreren Teilen ausgedruckt. Sie müssen wieder im Template die durch `@nnn` gekennzeichneten Lücken ausfüllen.

## Teil 1:

```
void moveWorm(struct board* aboard, struct worm* aworm,
enum GameStates* agame_state) {
    struct pos headpos;

    // Get the current position of the worm's head element
    headpos =  aworm->wormpos[aworm->headindex];
    // Compute new head position according to current heading.
    // Do not store the new head position in the array of positions, yet.
```

```
    headpos.x += aworm->dx;
    headpos.y += aworm->dy;

    // Check if we would hit something (for good or bad) or are going to leave
    // the display if we move the worm's head according to worm's
    // last direction.
    // We are not allowed to leave the display's window.
    if (headpos.x < 0) {
        *agame_state = WORM_OUT_OF_BOUNDS;
    } else if (headpos.x > getLastColOnBoard(aboard) ) {
        *agame_state = WORM_OUT_OF_BOUNDS;
    } else if (headpos.y < 0) {
        *agame_state = WORM_OUT_OF_BOUNDS;
    } else if (headpos.y > getLastRowOnBoard(aboard) ) {
        *agame_state = WORM_OUT_OF_BOUNDS;
    } else {
        // We will stay within bounds.
```

Im ersten Teil müssen Sie nur die rot markierten Stellen anpassen. Zum einen muss der neue Funktionsparameter `aboard` hinzugefügt werden. Zum anderen haben sich im Modul `board_model.c` die Namen der Zugriffsfunktionen (Getter) geändert, die die Grenzen des Spielfeldes zurückgeben.

## Teil 2: neue Kollisionsbehandlung

```
    // Check if the worms head hits any items at the new position on the board.
    // Hitting food is good, hitting barriers or worm elements is bad.
    switch ( getContentAt(aboard,headpos) ) {
        case BC_FOOD_1:
            *agame_state = WORM_GAME_ONGOING;
            // Grow worm according to food item digested
            growWorm(aworm, BONUS_1);
            decrementNumberOfFoodItems(aboard);
            break;
        case BC_FOOD_2:
            @001
        case BC_FOOD_3:
            @002
        case BC_BARRIER:
            // That's bad: stop game
            *agame_state = WORM_CRASH;
            break;
        case BC_USED_BY_WORM:
            // That's bad: stop game
            *agame_state = WORM_CROSSING;
            break;
        default:
            // Without default case we get a warning message.
            {;} // Do nothing. C syntax dictates some statement, here.
    }
}
```

Im mittleren Teil der Funktion moveWorm erfolgt die neue Kollisionsbehandlung. Wir lesen mittels

```
getContentAt(aboard,headpos)
```

aus, ob sich an der neuen Kopfposition ein Gegenstand befindet. Abhängig vom gespeicherten Code führen wir unterschiedliche Aktionen aus. Studieren Sie die Implementierung für den Fall `BC_FOOD_1` und ergänzen Sie dann die Lücken an den durch `@001` und `@002` gekennzeichneten Stellen.

### Teil 3: Analyse des Spielzustands (`*agame_state`)

```
    // Check the status of *agame_state
    // Go on if nothing bad happened
    if ( *agame_state == WORM_GAME_ONGOING ) {
        // So all is well: we did not hit anything bad and did not leave the
        // window. --> Update the worm structure.
        // Increment headindex
        // Go round if end of worm is reached (ring buffer)
        aworm->headindex++;
        if (aworm->headindex > aworm->cur_lastindex) {
            aworm->headindex = 0;
        }
        // Store new coordinates of head element in worm structure
        aworm->wormpos[aworm->headindex] = headpos;
    }
}
```

Im letzten, dritten Teil müssen Sie nur die Grenze `cur_lastindex` anstatt der bisherigen Grenze `maxindex` einsetzen (siehe rote Markierung).
Als letztes können Sie noch die komplette Funktion `isInUseByWorm` aus Ihrer Modul-Datei `worm_model.c` löschen. Diese Funktion wird nun nicht mehr benötigt.
Damit sind die Änderungen an der Modul-Datei `worm_model.c` abgeschlossen. Wenden wir uns nun der letzten verbleibenden Datei `worm.c` zu.

## Aufgabe 13 (Änderungen in der Funktion `initializeColors`)

Die Änderungen in der Funktion `initializeColors` sind minimal. Nehmen Sie Änderungen vor, so dass Ihr code folgendem Ausschnitt entspricht:

```
// Initialize colors of the game
void initializeColors() {
    // Define colors of the game
    start_color();
    init_pair(COLP_USER_WORM,     COLOR_GREEN,    COLOR_BLACK);
    init_pair(COLP_FREE_CELL,     COLOR_BLACK,    COLOR_BLACK);
    init_pair(COLP_FOOD_1,        COLOR_YELLOW,   COLOR_BLACK);
    init_pair(COLP_FOOD_2,        COLOR_MAGENTA,  COLOR_BLACK);
    init_pair(COLP_FOOD_3,        COLOR_CYAN,     COLOR_BLACK);
    init_pair(COLP_BARRIER,       COLOR_RED,      COLOR_BLACK);
}
```

## Aufgabe 14 (Änderungen in der Funktion `readUserInput`)

Die Änderungen in der Funktion `readUserInput` sind ebenfalls minimal. Eigentlich müsste man an der Funktion gar nichts ändern, aber zu Debug-Zwecken soll ein Cheat-Modus eingefügt werden. Wenn der Benutzer die Taste ‚**g**'drückt, soll der Wurm so wachsen, als ob er einen Futterbrocken der Kategorie 3 gefressen hätte[4].
Im Folgenden ist nur der Code-Ausschnitt gezeigt, den Sie der Funktion `readUserInput` an geeigneter Stelle hinzufügen müssen:

[4]Falls bei Ihnen die Taste ‚**g**'schon belegt ist, können Sie jede beliebige andere Taste verwenden.

```
case 'g' : // For development: let the worm grow by BONUS_3 elements
        growWorm(aworm, BONUS_3);
        break;
```

# Aufgabe 15 (Änderungen in der Funktion `doLevel`)

Nun kommen wir zur Funktion `doLevel`, die bekanntlich die Hauptschleife des Spiels implementiert.
Die von Ihnen auszuführenden Änderungen sind im Folgenden durch Codefragmente der Funktion `doLevel` skizziert. Der Code wird in drei Teilen präsentiert, Änderungen sind rot hervorgehoben:

## Teil 1: Initialisierung

```
enum ResCodes doLevel() {
    struct worm userworm;  // Currently we just use one user worm in the game
    struct board theboard; // Our game board
    enum GameStates game_state; // The current game_state
    ...
    game_state = WORM_GAME_ONGOING;

    // Setup the board
    res_code = initializeBoard(&theboard);
    if ( res_code != RES_OK) {
        return res_code;
    }
    // Initialize the current level
    res_code = initializeLevel(&theboard);
    if ( res_code != RES_OK) {
        return res_code;
    }
    ...
    bottomLeft.y =  getLastRowOnBoard(&theboard);
    bottomLeft.x =  0;

    res_code = initializeWorm(&userworm, WORM_LENGTH, WORM_INITIAL_LENGTH,
    bottomLeft , WORM_RIGHT, COLP_USER_WORM);
    if ( res_code != RES_OK) {
        return res_code;
    }
    // Hier ueberfluessigen Aufruf von showBorderLine() loeschen
    // Show worm at its initial position
    showWorm(&theboard, &userworm);
```

Eine Neuerung im ersten Teil ist die Definition der Strukturvariablen `theboard` vom Typ **`struct board`**. In dieser Strukturvariablen speichern wir die Belegung des Spielbretts. Die Variable wird an andere Funktionen jeweils per Adresse übergeben. Danach initialisieren wir das Spielbrett mit `initializeBoard` und belegen es durch `initializeLevel` mit Gegenständen. Die Initialisierungsfunktion für den Wurm wird mit einem neuen Parameter für die initiale Länge des Wurms `WORM_INITIAL_LENGTH` aufgerufen.

## 0.1 Teil 2: Hauptschleife des Spiels

```
while(!end_level_loop) {
    . . .
    cleanWormTail(&theboard, &userworm);
```

```
    . . .
    moveWorm(&theboard, &userworm, &game_state);
    . . .
    showWorm(&theboard, &userworm);
    . . .
    showStatus(&theboard, &userworm);
    . . .
    // Display all the updates
    refresh();
    // Are we done with that level?
    if (getNumberOfFoodItems(&theboard) == 0) {
        end_level_loop = true;
    }
    // Start next iteration
```

In der Hauptschleife werden diverse Unterfunktionen mit dem neuen Parameter `&theboard` aufgerufen (siehe dazu die den Modulen entsprechenden Header-Dateien). Bevor wir einen neuen Schleifendurchlauf ausführen, prüfen wir, ob überhaupt noch Futterbrocken auf dem Spielfeld vorhanden sind.

### Teil 3: Behandlung des Abbruchs nach der Hauptschleife

```
// For some reason we left the control loop of the current level
// Check why according to game_state
switch (game_state) {
    case WORM_GAME_ONGOING:
    if (getNumberOfFoodItems(&theboard) == 0) {
        showDialog("Sie habe diese Runde erfolgreich beendet !!!",
        "Bitte Taste druecken");
    } else {
        showDialog("Interner Fehler!","Bitte Taste druecken");
        // Correct result code
        res_code = RES_INTERNAL_ERROR;
    }
    break;
    case WORM_GAME_QUIT:
    // User must have typed 'q' for quit
    showDialog("Sie haben die aktuelle Runde abgebrochen!",
    "Bitte Taste druecken");
    break;
    case WORM_CRASH:
    showDialog("Sie haben das Spiel verloren,"
    " weil Sie in die Barriere gefahren sind",
    "Bitte Taste druecken");
    break;
    case WORM_OUT_OF_BOUNDS:
    ...
```

Im dritten Teil der Funktion `doLevel` untersuchen wir in einer Fallunterscheidung, weshalb die Hauptschleife verlassen wurde. Hier sind zwei neue Fälle hinzugekommen.
Damit sind nun auch die Änderungen an der Datei `worm.c` abgeschlossen.

## Endkontrolle

Ihr Projekt `Worm070` sollte nun wieder in einem übersetzbaren Zustand sein. Testen Sie diese Annahme durch den folgenden Befehl:
`$ make clean; make`
Das Programm sollte sich ohne Fehler kompilieren lassen, und auch die Ausführung der Binärdatei

`bin/worm` sollte fehlerfrei möglich sein. Falls das bei Ihnen nicht so ist, ...

Zum Abschluss speichern Sie die Quellen Ihrer finalen lauffähigen Version im Repository.
Zum Beispiel so:

```
$ git status
$ git add .
$ git commit -m "Worm070 final"
$ git push
```

# Schlußbemerkung

Im Rahmen dieses Aufgabenblatts haben wir eine neue Datenstruktur **`struct board`** eingeführt, mit deren Hilfe wir speichern, welche Gegenstände sich auf dem Spielbrett befinden. Zu den Gegenständen zählen hierbei Futterbrocken, Hindernisse und auch Wurmelemente.
Die Verwendung der neuen Datenstruktur erlaubt uns unter anderem eine verbesserte Kollisionsbehandlung, da wir nun nicht nur Kollisionen des Wurms mit sich selbst entdecken können.
Desweiteren haben wir implementiert, dass der Wurm nach dem Vertilgen eines Futterbrockens wächst.
Das Kernstück der Datenstruktur **`struct board`** ist ein zweidimensionales Array, in dem wir die Belegung der Zellen des Spielbretts durch Gegenstände mit Hilfe spezieller Codes vom Typ **`enum`** `BoardCodes` speichern.
Es ist von großer Wichtigkeit, dass Sie alle durchgeführten Teilschritte verstehen. Studieren Sie insbesondere die Definition der Struktur **`struct board`** und die Übergabe der Strukturvariable `theworm` per Adresse an andere Funktionen.

## Abnahme der Vorführaufgabe

Ihr Kursbetreuer wird sich von Ihnen zum Zweck der Lern- und Erfolgskontrolle das Programm vorführen lassen. Rechnen Sie damit, dass Sie

- Ihrem Betreuer den Zweck einzelner Anweisungen im Programm erklären müssen
- auf Anfrage individuelle Änderungen vornehmen und erklären müssen
- zeigen müssen, dass Sie den Mikrozyklus `Edit`/`Compile`/`Run` beherrschen
- Ihren C-Code sauber formatiert haben (einheitlich Einrücken) und an geeigneten Stellen Ihren Code auch sinnvoll dokumentieren
- in der Lage sind, den C-Code in den dafür vorgesehenen Unterverzeichnissen ordentlich zu organisieren.
- in der Lage sind, ihr `git`-Repository bei *bitbucket* zu verwalten und den sicheren Umgang mit den Befehlen `git status/add/commit/clone/push/pull` beherrschen.

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