Technische Hochschule Ingolstadt
Fakultät für Informatik

Praktikum zu „Grundlagen der Programmierung“

Blatt 7

Lernziele: Verwendung von Strukturen (Verbunden)
Erforderliche Kenntnisse: Inhalte aus der Vorlesung zum Thema Strukturen
Voraussetzungen:

- 1. Vollständige Bearbeitung des letzten Blattes 06 (`Worm030`).

Übersicht

Dieses Aufgabenblatt widmet sich dem Thema der Strukturierung und Kapselung von Daten mittels *Strukturen* (Verbunden), die in C durch das Schlüsselwort **`struct`** definiert werden können. In Verbunden können einzelne Variablen, die erst gemeinsam eine Sinneinheit bilden, zu einer strukturierten Variablen zusammengefasst werden.
Der Vorteil dieser Zusammenfassung besteht unter anderem darin, dass Strukturen als Ganzes per Wertkopie an eine Funktion übergeben werden können und ebenso als Ganzes das Resultat eines Funktionsaufrufs sein können.
Natürlich können Strukturen auch per Adresse an eine Funktion übergeben oder von dieser zurückgegeben werden, wenn man das Kopieren der ganzen Struktur bei der Übergabe vermeiden will.
Im Rahmen dieses Aufgabenblatts werden wir zwei Strukturen (Verbunde) einführen:
1. Eine Struktur **`struct pos`** für die Speicherung von Koordinatenpaaren (Zeile, Spalte). Bisher haben wir immer mit zwei getrennten Variablen für Zeile und Spalte gearbeitet. 2. Eine Struktur **`struct worm`**, die alle Daten beinhaltet, die zur Speicherung eines Wurms benötigt werden. Bisher haben wir hierfür diverse globale Variable wie `theworm_headindex`, `theworm_dx` etc. verwendet. Dazu zählen auch die bisher getrennten Arrays für die Koordinaten der einzelnen Wurmelemente `theworm_wormpos_y` und `theworm_wormpos_x`. An deren Stelle tritt nun ein einziges Feld von Koordinatenpaaren, wobei die bereits erwähnte neue Strutktur **`struct pos`** zum Tragen kommt. Zusätzlich zur Einführung von Strukturen werden wir die Darstellung des Spiels verbessern. Wir reservieren mehrere Zeilen am unteren Rand des Fensters, eine sognannte *Message Area*, und nützen diesen Bereich zur Ausgabe von Statusinformationen und für Fehlermeldungen.

Aufgabe 1 (Kopieren, Auspacken und Umbenennen des Templates)

Im Moodle-Kursraum finden Sie im Verzeichnis `Praktikum` die Datei `Worm050Template.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
```

Vergewissern Sie sich mittels des Befehls `ls`, dass die Zip-Datei im Verzeichnis vorhanden ist.

`$ ls Worm050Template.zip`
Entpacken Sie die Zip-Datei durch den nachfolgenden Befehl. Dadurch wird das Verzeichnis `Worm050Template` mit einigen Dateien darin angelegt.
`$ unzip Worm050Template.zip`
Listen Sie den Inhalt des neuen Unterverzeichnisses.
`$ ls -la Worm050Template`
Benennen Sie das Verzeichnis `Worm050Template` um in `Worm050`.
`$ mv Worm050Template Worm050`
Listen Sie den Inhalt des umbenannten Unterverzeichnisses.
`$ ls -la Worm050`
Listen Sie den Inhalt des aktuellen Verzeichnisses.
`$ ls -la`
Löschen Sie das Zip-Archiv.
`$ rm -f Worm050Template.zip`
Führen Sie den nachfolgenden Kopierbefehl[1] aus. Dadurch werden die sechs Dateien

- `board_model.c` und `board_model.h`
- `worm_model.c` und `worm_model.h`
- `worm.c` und `worm.h`

aus dem Verzeichnis `Worm030` in das neue Verzeichnis `Worm050` übernommen (kopiert). Diese Dateien bilden die Ausgangssituation für die Neuerungen in der Version `Worm050`.
`$ cp Worm030/board_model.{c,h} Worm030/worm*.{c,h} Worm050`

Aufgabe 2 (Test der Ausgangssituation)

Öffnen Sie eine Shell und wechseln Sie in dieser Shell ins Verzeichnis
`~/GdP1/Praktikum/Code/Worm050`.
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 `Worm030` kopierten sechs Dateien (rot) sind alle anderen Dateien durch das Auspacken des Templates `Worm050Template.zip` entstanden. Im Vergleich zur Vorversion `Worm030` hat sich das `Makefile` geringfügig geändert. Die neu hinzu gekommenen Dateien `messages.c` und `messages.h` werden im `Makefile` zusätzlich aufgeführt. Diesen beiden Dateien wenden wir uns erst in einer der letzten Teilaufgaben dieses Blatts zu.
In der nun hergestellten Ausgangssituation sollte sich das gesamte Programm fehlerfrei übersetzen lassen. Überprüfen Sie diese Annahme durch Ausführung des folgenden Befehls:
`$ make clean; make`
Die Ausführung der Binärdatei `bin/worm` sollte ebenfalls fehlerfrei möglich sein.

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

Zum Abschluss dieser Teilaufgabe speichern Sie die Quellen Ihrer initialen lauffähigen Version im Repository. Zum Beispiel so:

```
$ git status
$ git add .
$ git commit -m "Worm050 initial"
```

Aufgabe 3 (Definition der Struktur `struct pos`)

Im ersten Schritt definieren wir die neue Struktur **`struct pos`**, die von nun an die Koordinaten für die Position (Zeile, Spalte) eines Symbols auf dem Spielfeld modelliert.
Öffnen Sie die Datei `board_model.h` mit einem Editor und fügen Sie die folgende Definition des Datentyps **`struct pos`** hinzu.
Ein guter Platz für die Definition wäre zum Beispiel direkt nach der letzten `#include`-Direktive.

```
// Positions on the board
struct pos {
        int y;    // y-coordinate (row)
        int x;    // x-coordinate (column)
};
```

Speichern Sie die Änderungen in der Datei `board_model.h`.

Aufgabe 4 (Definition der Struktur `struct worm`)

Nun legen wir den Grundstein für die wesentliche Neuerung der Version `Worm050`. Wir führen die neue Datenstruktur **`struct worm`** ein, die alle Datenaspekte des Wurms modelliert.
Öffnen Sie die Datei `worm_model.h` und fügen Sie die nachfolgende Strukturdefinition nach der letzten `#include`-Direktive ein (also ziemlich am Anfang der Header-Datei).

```
// A worm structure
struct worm
{
        int maxindex;    // Last usable index into the array pointed to by wormpos
        int headindex;   // An index into the array for the worm's head position
        // 0 <= headindex <= maxindex

        struct pos wormpos[WORM_LENGTH]; // Array of x,y positions of all elements
        // of the worm

        // The current heading of the worm
        // These are offsets from the set {-1,0,+1}
        int dx;
        int dy;
        // Color of the worm
        enum ColorPairs wcolor;
};
```

Beim Studium dieser Verbund-Definition werden Sie feststellen, dass die Struktur **`struct worm`** alle vormals durch mehrere globale Variablen modellierten Datenaspekte des Wurms als Komponenten enthält.
Insbesondere werden die vormals getrennten Arrays `theworm_wormpos_y[WORM_LENGTH]` und

`theworm_wormpos_x[WORM_LENGTH]` nun als ein Array `wormpos[WORM_LENGTH]` von Elementen des Typs **`struct pos`** ersetzt.
Durch die Verwendung der Struktur **`struct pos`** können wir endlich die Positionskoordinaten eines Symbols auf dem Spielfeld als Einheit behandeln, was viel natürlicher erscheint als die künstliche Trennung der Koordinaten in zwei unterschiedlichen Arrays.
Da wir in der Strukturdefinition von **`struct worm`** auf die Struktur **`struct pos`** Bezug nehmen (siehe rote Markierung), muss diese dem Compiler vorher bekannt gemacht werden. Das erreichen wir, in dem wir am Anfang der Header-Datei `worm_model.h` die `CPP`-Direktive

```
#include "board_model.h"
```

hinzufügen.
Dadurch wird die Definition der Struktur **`struct pos`** rechtzeitig vor ihrer Verwendung geladen.
Die Einführung der beiden Strukturen **`struct pos`** und **`struct worm`** zieht weitreichende Veränderungen nach sich. Die konsequente Verwendung dieser neuen Strukturen im restlichen Code wird Gegenstand der noch folgenden Aufgaben sein.

Vorbetrachtungen (Ersetzung globaler Variablen durch lokale Variablen)

Neben der Einführung von strukturierten Datentypen wollen wir im Rahmen dieses Aufgabenblatts auch vollständig auf die Verwendung von globalen Variablen verzichten.
Globale Variablen sind auf den ersten Blick sehr bequem, da auf sie aus allen Funktionen heraus direkt zugegriffen werden kann. Der Preis für die Verwendung von globalen Variablen ist aber immer eine starre und unflexible Implementierung all derer Funktionen, die diese globalen Variablen verwenden.
Eine Funktion, die explizit auf eine globale Variable zugreift, ist syntaktisch an den Namen dieser globalen Variablen gebunden. Wird hingegen der Wert oder die Adresse einer Variablen als Parameter an die Funktion übergeben, so kann die Funktion problemlos mehrfach mit unterschiedlichen Argumenten, also auch mit unterschiedlichen Variablen, aufgerufen werden.
Führen wir uns diesen Sachverhalt am Beispiel der bisher verwendeten globalen Variablen vor Augen, die alle den Präfix `theworm_` tragen. Diese Variablen zusammen haben bisher die verteilte Datenstruktur des Wurms gebildet. Mehrere Funktionen, darunter zum Beispiel `moveWorm`, greifen derzeit direkt auf diese Variablen zu. In der jetzigen Implementierung kann `moveWorm` nur den einen Benutzerwurm manipulieren, da die Namen der globalen Variablen explizit in der Funktion erwähnt werden. Das ist unflexibel!
Stattdessen wollen wir von nun an nur noch eine **Strukturvariable** `userworm` vom Typ **`struct worm`** für die Modellierung des Benutzerwurms verwenden, welche zudem als **lokale Variable** in der Funktion `doLevel` der Datei `worm.c` definiert sein soll.
Zunächst scheint die Einführung einer lokalen Variable für den Benutzerwurm Nachteile zu bringen, denn aufgrund der lokalen Definition kann damit eine andere Funktion nur noch auf die Datenstruktur des Wurms zugreifen, wenn ihr der Wurm als Parameter übergeben wird. Wir müssen also bei all diesen Funktionen den Benutzerwurm als Parameter einführen.
Auf den zweiten Blick erkennen wir jedoch das enorme Potential hinter dieser Umstellung. Indem wir einen Wurm (nicht den Wurm) als Parameter übergeben, können unsere Funktionen in Zukunft beliebige Würmer manipulieren. Sie sind nicht mehr per Zugriff über den festen Namen einer globalen Variablen auf den einen Benutzerwurm beschränkt. Wir werden in einer späteren

Version des Spiels (ab `Worm090`) ausgiebig Gebrauch von diesem Potential machen, wenn wir eine beliebige Anzahl automatisch bewegter Systemwürmer durch unsere Funktionen manipulieren lassen. So ist es später, aufgrund des neuen Wurm-Parameters, unserer Funktion `moveWorm` egal, ob sie nun gerade den Benutzerwurm oder einen der vielen Systemwürmer einen Schritt auf dem Spielfeld weiter bewegt. Das gleiche gilt für die anderen Funktionen, die irgendeinen Aspekt eines Wurms manipulieren oder auslesen.

Damit die gerade skizzierte Idee umgesetzt werden kann, müssen wir in alle Funktionen, die Aspekte des bzw. eines Wurms manipulieren, den besprochenen zusätzlichen Parameter für den zu manipulierenden Wurm hinzufügen.
Zum einen aus Gründen der Effizienz, zum anderen aber auch, damit Änderungen an der Wurmstruktur durchgeführt werden können (Seiteneffekt), werden wir den Wurm immer **per Adresse** übergeben.

Aufgabe 5 (Datei `worm_model.h`: zusätzlicher Parameter `struct worm aworm`)

Beginnen wir den Einbau des zusätzlichen Parameters in der Datei `worm_model.h`.
Diese Header-Datei enthält neben den Definitionen mehrerer Datenstrukturen auch noch alle Vorwärtsdeklarationen des Moduls `worm_model.c`. In diesem Modul sind gerade alle Funktionen zusammengefasst, die irgendwelche Manipulationen an dem (an einem) Wurm vornehmen.
Alle diese Funktionen erhalten nun einen zusätzlichen Parameter `struct worm* aworm`. Die Verwendung des Variablennamens `aworm` soll verdeutlichen, dass es sich um irgendeinen beliebigen Wurm handeln kann, dessen Adresse hier übergeben wird.
Fügen Sie, wie nachfolgend gezeigt, in allen Vorwärtsdeklarationen der Datei `worm_model.h` den zusätzlichen Parameter ein. An zwei Stellen ergeben sich zudem Änderungen aufgrund der Einführung des Verbunds `struct pos`:

```
extern enum ResCodes initializeWorm(struct worm* aworm, int len_max,
struct pos headpos, enum WormHeading dir, enum ColorPairs color);
extern void showWorm(struct worm* aworm);
extern void cleanWormTail(struct worm* aworm);
extern void moveWorm(struct worm* aworm, enum GameStates* agame_state);
extern bool isInUseByWorm(struct worm* aworm, struct pos new_headpos);
extern void setWormHeading(struct worm* aworm, enum WormHeading dir);
```

Aufgabe 6 (Änderungen im Modul `worm_model.c`)

Die Einführung der Datenstrukturen `struct worm` und `struct pos`, sowie der geplante Verzicht auf globale Variablen und stattdessen Einführung eines neuen Parameters `struct worm* aworm` resultieren zwangsläufig in mehreren Änderungen im Modul `worm_model.c`.
Öffnen Sie die Modul-Datei `worm_model.c` in einem Editor und führen Sie die nachfolgend beschriebenen Änderungen aus.
Löschen Sie zuerst alle Definitionen von globalen Variablen in der Modul-Datei, d.h. löschen Sie alle Definitionen von Variablen, die den Präfix `theworm_` haben. Diese sind

```
theworm_maxindex
theworm_headindex
theworm_wormpos_x
theworm_wormpos_y
theworm_dx
theworm_dy
theworm_wcolor
```

Fügen Sie sodann, analog zu den Vorwärtsdeklarationen der Datei `worm_model.h`, in alle Funktionen des Moduls den neuen Parameter **`struct worm*`** `aworm` ein. Vergessen Sie dabei auch nicht bei den Funktionen `initializeWorm` und `isInUseByWorm` den Parameter vom Typ **`struct pos`**.
In den Rümpfen der Funktionen müssen Sie alle bisherigen Zugriffe auf die globalen Variablen mit Präfix `theworm_` durch entsprechende Zugriffe über den neuen Parameter `aworm` ersetzen. Da es sich bei `aworm` um eine Parametervariable handelt, die die Adresse einer **`struct worm`** enthält, müssen Sie beim Zugriff zunächst die Adresse de-referenzieren und dann auf die jeweilige Komponente der Struktur zugreifen.
Aus der Vorlesung wissen Sie, dass zum Beispiel beim Zugriff auf die Komponente `headindex` statt des umständlichen zweistufigen Zugriffs

- Dereferenz der Adresse mittels Stern-Operator: `*aworm`
- Danach Zugriff auf Komponente mittels Punkt-Operator: `(*aworm).headindex`

die wesentlich kompaktere Schreibweise mittels Pfeil-Operator verwendet werden kann:

```
aworm -> headindex
```

Um Ihnen den Einstieg zu erleichtern, ist nachfolgend der vollständig angepasste Code der Funktion `initializeWorm` abgedruckt.

```
enum ResCodes initializeWorm(struct worm* aworm, int len_max,
struct pos headpos, enum WormHeading dir, enum ColorPairs color) {
        int i;
        // Initialize last usable index to len_max -1
        aworm->maxindex = len_max - 1 ;
        // Initialize headindex
        aworm->headindex = 0;   // Index pointing to head position is set to 0
        // Mark all elements as unused in the array of positions.
        // This allows for the effect that the worm appears element by element at
        // the start of each level
        for (i = 0; i <= aworm->maxindex; i++) {
                aworm->wormpos[i].y = UNUSED_POS_ELEM;
                aworm->wormpos[i].x = UNUSED_POS_ELEM;
        }
        // Initialize position of worms head
        aworm->wormpos[aworm->headindex] = headpos;
        // Initialize the heading of the worm
        setWormHeading(aworm, dir);
        // Initialze color of the worm
        aworm->wcolor = color;
        return RES_OK;
}
```

Ändern Sie nun analog zu obigem Beispiel in allen anderen Funktionen des Moduls den Zugriff auf den neuen Parameter `aworm`. Vergessen Sie auch bitte nicht, dass Sie beim Zugriff auf die Koordinaten einer Position nun über eine Struktur des Typs **`struct pos`** zugreifen müssen bzw. können. Diese Strukturen können zum Beispiel auch als Ganzes zugewiesen oder als Parameter übergeben werden (z.B. in der Funktion `isInUseByWorm` oder im obigen Beispiel die Zuweisung von `headpos`).

Aufgabe 7 (Änderungen im Modul `worm.c`)

Nun wenden wir uns der Modul-Datei `worm.c` zu. Da wir die globalen Variablen für die Datenstruktur des Benutzerwurms eliminiert haben und stattdessen eine lokale Variable von Typ **`struct worm`** einführen wollen, muss diese lokale Variable nun in irgendeiner Funktion definiert werden.
Ein guter Platz für die Definition der Strukturvariable des Benutzerwurms ist die Funktion `doLevel`, da hier die Initialisierung des Wurms (`initializeWorm`) aufgerufen wird und auch die meisten anderen Funktionen zur Manipulation des Wurms aufgerufen werden: (`cleanWormTail`, `moveWorm`, `showWorm`).
Nachfolgend ist der Code der Funktion `doLevel` in Ausschnitten gezeigt, wobei alle geänderten Zeilen aufgeführt sind:

```
enum ResCodes doLevel() {
        struct worm userworm;  // Local variable for storing the user's worm
        enum GameStates game_state; // The current game_state
        ...
        struct pos bottomLeft;
        ...
        // Initialize the userworm with its size, position, heading.
        bottomLeft.y =  getLastRow();
        bottomLeft.x =  0;

        res_code = initializeWorm(&userworm, WORM_LENGTH, bottomLeft,
        WORM_RIGHT, COLP_USER_WORM);
        ...
        // Show worm at its initial position
        showWorm(&userworm);
        ...
        while(!end_level_loop) {
                // Process optional user input
                readUserInput(&userworm, &game_state);
                if ( game_state == WORM_GAME_QUIT ) {
                        ...
                }
                // Process userworm
                // Clean the tail of the worm
                cleanWormTail(&userworm);
                // Now move the worm for one step
                moveWorm(&userworm, &game_state);
                ...
                // Show the worm at its new position
                showWorm(&userworm);
                ...
```

Die Änderungen zeigen, dass wir die lokal in `doLevel` definierte Strukturvariable `userworm` an alle Funktionen, die auf den Wurm zugreifen müssen, per Adresse (`&userworm`) übergeben. Bis auf eine Ausnahme sind das alles Funktionen, die wir bereits im Modul `worm_model.c` geändert haben. Beim Betrachten der Änderungen fällt aber auf, dass auch der Aufruf von `readUserInput` um das Argument `&userworm` erweitert wurde. Der Grund hierfür liegt darin, dass innerhalb von `readUserInput` die Funktion `setWormHeading` aufgerufen wird. Diese benötigt nun ebenfalls ein Argument vom Typ **`struct worm`**`*`, welches wir durch die Funktion `readUserInput` durchschleusen müssen. Ändern Sie also nun noch die Implementierung der Funktion `readUserInput`, in dem Sie den neuen Parameter **`struct worm`**`* aworm` hinzufügen und wie skizziert in den Aufrufen der Funktion `setWormHeading` verwenden. Ihr Projekt im Vezeichnis `Worm050` sollte nun wieder übersetzbar sein. Prüfen Sie diese Annahme durch:

```
$ 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, haben Sie nicht alle bisher angegebenen Schritte korrekt ausgeführt.

Vorbetrachtungen: (Hinzufügen einer Message Area)

Wir wenden uns nun einem neuen Thema zu. Um die Darstellung des Spiels zu verbessern, reservieren wir am unteren Rand des Spielfelds vier Zeilen für eine sogenannte *Message Area*. Die erste der vier Zeilen enthält eine Trennlinie, die der Wurm nicht überschreiten darf. Die restlichen Zeilen werden zur Ausgabe von Status- und Fehlermeldungen benutzt.
Die nachfolgenden beiden Bilder zeigen das Spielfeld mit der neuen Message Area:

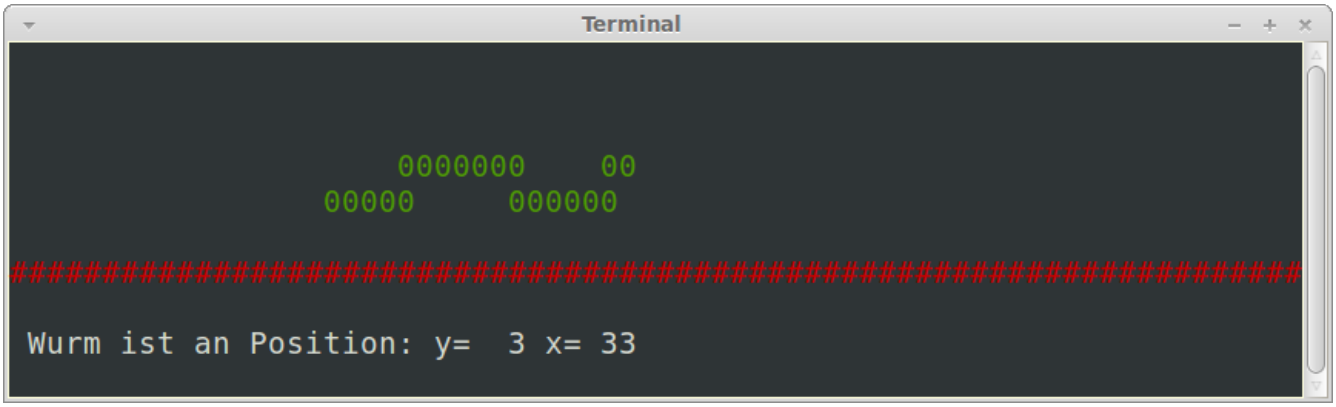

Abbildung 1: Die Message Area zeigt Informationen zur aktuellen Kopfposition an.

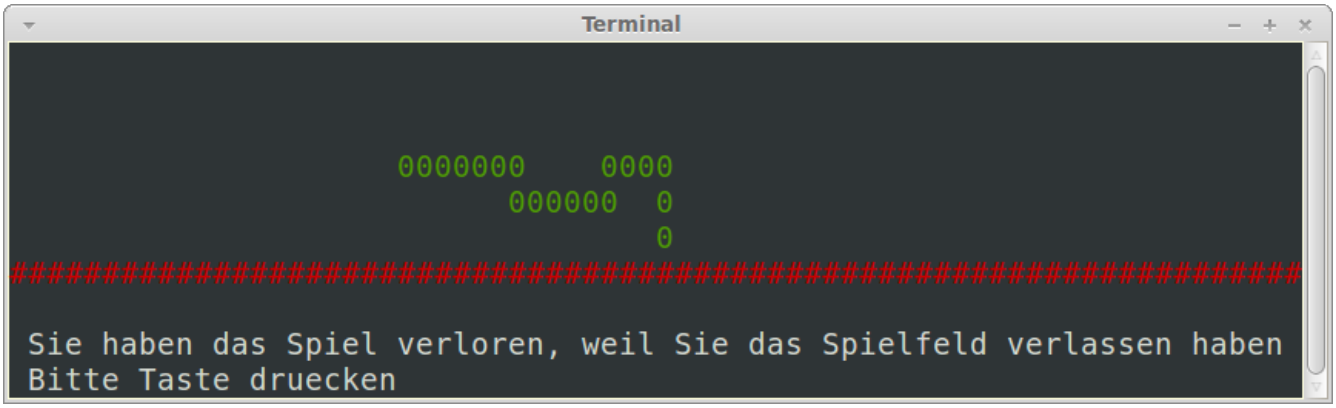

Abbildung 2: Die Message Area zeigt Informationen zu einem aufgetreten Fahrfehler an.

In Abb. 1 wird die Message Area zur Ausgabe von Informationen über die Kopfposition des Wurms benutzt, wohingegen in Abb. 2 der Benutzer auf seinen Fahrfehler und den Grund des Spielabbruchs informiert wird.
Die Funktionen, die die Ausgabe in die Message Area ausführen, werden Ihnen bereits fertig in

den Dateien `messages.h` und `messages.c` geliefert. In den folgenden Teilaufgaben müssen Sie lediglich den Aufruf der Funktionen und die Konfiguration der Message-Area implementieren.

Aufgabe 8 (Aktivierung des Moduls `messages.c` im `Makefile`)

Zuerst müssen Sie die Übersetzung und das Binden des Moduls `messages.c` im `Makefile` aktivieren.
Öffnen Sie das Makefile und entfernen Sie die Kommentarzeichen „#“in den Zeilen 16 und 23.
Jetzt integriert das Makefile auch das Modul `messages.c`.
Im gegenwärtigen Zustand ist das Modul aber noch nicht übersetzbar. Wir müssen zunächst in den restlichen Dateien noch Änderungen vornehmen.

Aufgabe 9 (Änderungen an `worm.h` für Integration der Message Area)

In der Header-Datei `worm.h` sind ein paar kleine Änderungen vorzunehmen. Im Wesentlichen müssen wir die Anzahl der für die Message Area zu reservierenden Zeilen hinterlegen (`ROWS_RESERVED`) und das Symbol für den Trennstrich zwischen Spielbereich und Message Area sowie seine Farbe definieren (`BARRIER`). Desweiteren legen wir die Breite des Spielfeldes (`MIN_NUMBER_OF_COLS`) neu fest und definieren einen weiteren Fehlercode für das Spiel (`RES_INTERNAL_ERROR`).
Führen Sie die Änderungen in der Datei `worm.h` wie im Folgenden skizziert durch:

```
// Result codes of functions
enum ResCodes {
        RES_OK,
        RES_FAILED,
        RES_INTERNAL_ERROR,
};
...
// Dimensions and bounds
#define NAP_TIME    100   // Time in milliseconds  . . .
#define ROWS_RESERVED 4   // Lines reserved for the message area
#define MIN_NUMBER_OF_ROWS  3  // The guaranteed number of . . .
#define MIN_NUMBER_OF_COLS 70  // The guaranteed number of columns . . .
...
enum ColorPairs {
        COLP_USER_WORM = 1,
        COLP_FREE_CELL,
        COLP_BARRIER,
};
...
#define SYMBOL_FREE_CELL  ' '
#define SYMBOL_BARRIER '#'
#define SYMBOL_WORM_INNER_ELEMENT '0'
...
```

Aufgabe 10 (Änderungen an `board_model.c` für Integration der Message Area)

Die Änderungen in dieser Datei sind minimal. Wir ändern lediglich die Getter-Funktion `getLastRow` so ab, dass die Berechnung der Anzahl der zur Verfügung stehenden Zeilen die Reservierung der Zeilen für die Message Area reflektiert.

```
// Get the last usable row on the display
int getLastRow() {
        return LINES - 1 - ROWS_RESERVED;
}
```

Aufgabe 11 (Änderungen an `worm_model.c,h` für Integration der Message Area)

In der Message Area wollen wir während des Spiels die Position (Koordinaten) des Wurmkopfs ausgeben. Dazu muss der Code im Modul `messages.c` in der Lage sein, die Position des Kopfs zu erfragen. Zu diesem Zweck führen wir eine Getter-Funktion `getWormHeadPos` im Modul `worm_model.c` ein.
Fügen Sie folgende Funktionsdefinition in der Datei `worm_model.c` hinzu:

```
// Getters
struct pos getWormHeadPos(struct worm* aworm) {
        // Structures are passed by value!
        // -> we return a copy here
        return aworm->wormpos[aworm->headindex];
}
```

Fügen Sie der Header-Datei `worm_model.h` eine entsprechende Vorwärtsdeklaration der Funktion hinzu.

Aufgabe 12 (Ansteuerung der Message Area aus `worm.c`)

Als letztes müssen wir nun (im Wesentlichen) nur noch die Funktionen des Moduls `messages.c` aus der Hauptschleife des Spiels in der Funktion doLevel ansteuern.
Führen Sie die im Folgenden skizzierten Änderungen an der Datei `worm.c` durch:

```
#include "prep.h"
#include "messages.h"
#include "worm.h"
...
void initializeColors() {
        // Define colors of the game
        ...
        init_pair(COLP_FREE_CELL,     COLOR_BLACK,    COLOR_BLACK);
        init_pair(COLP_BARRIER,       COLOR_RED,      COLOR_BLACK);
}
...
enum ResCodes doLevel() {
        ...
        // Show border line in order to separate the message area
        showBorderLine();
        // Show worm at its initial position
```

```
        showWorm(&userworm);
        ...
        while(!end_level_loop) {
                ...
                // Show the worm at its new position
                showWorm(&userworm);
                // END process userworm
                // Inform user about position and length of userworm in status window
                showStatus(&userworm);
                // Sleep a bit before we show the updated window
                napms(NAP_TIME);
        }
        // Preset res_code for rest of the function
        res_code = RES_OK;
        // For some reason we left the control loop of the current level
        // Check why according to game_state
        switch (game_state) {
                case WORM_GAME_QUIT:
                // User must have typed 'q' for quit
                showDialog("Sie haben die aktuelle Runde abgebrochen!",
                "Bitte Taste druecken");
                break;
                case WORM_OUT_OF_BOUNDS:
                showDialog("Sie haben das Spiel verloren,"
                " weil Sie das Spielfeld verlassen haben",
                "Bitte Taste druecken");
                break;
                case WORM_CROSSING:
                showDialog("Sie haben das Spiel verloren,"
                " weil Sie einen Wurm gekreuzt haben",
                "Bitte Taste druecken");
                break;
                default:
                showDialog("Interner Fehler!","Bitte Taste druecken");
                // Set error result code. This should never happen.
                res_code = RES_INTERNAL_ERROR;
        }
        // Normal exit point
        return res_code;
} // End of function doLevel
...
int main(void) {
        enum ResCodes res_code;          // Result code from functions
        // Here we start
        initializeCursesApplication();  // Init various settings of our application
        initializeColors();             // Init colors used in the game

        // Maximal LINES and COLS are set by curses for the current window size.
        // Note: we do not cope with resizing in this simple examples!

        // Check if the window is large enough to display messages in message area
        // and whether it has space for at least MIN_NUMBER_OF_ROWS lines for
        // the worm
        if (LINES < ROWS_RESERVED + MIN_NUMBER_OF_ROWS || COLS < MIN_NUMBER_OF_COLS)
        {
                // Since we not even have the space for displaying messages
                // we print a conventional error message via printf after
                // the call of cleanupCursesApp()
                cleanupCursesApp();
                printf("Das Fenster ist zu klein: wir brauchen mindestens %dx%d\n",
                MIN_NUMBER_OF_COLS, MIN_NUMBER_OF_ROWS + ROWS_RESERVED );
                res_code = RES_FAILED;
        } else {
                res_code = doLevel();
                cleanupCursesApp();
        }
        return res_code;
```

```
}
```

Endkontrolle

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

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

Schlußbemerkung

Im Rahmen dieses Aufgabenblatts haben wir eine Reihe globaler Variablen, die bisher in Summe die Datenstruktur des Wurms gebildet haben, eliminiert und durch eine strukturierte Variable vom Typ **`struct worm`** ersetzt. Diese Variable wird jetzt lokal in der Funktion `doLevel` definiert und als Parameter per Adresse an alle Funktionen übergeben, die auf die Datenstruktur des Wurms zugreifen müssen.
Dadurch haben wir unsere Funktionen flexibler gemacht und den Grundstein für die spätere gleichzeitige Darstellung beliebig vieler Würmer geschaffen.
Desweiteren haben wir eine Message Area eingeführt, in der wir Statusanzeigen und Fehlermeldungen darstellen können. Es ist von großer Wichtigkeit, dass Sie alle durchgeführten Teilschritte verstehen. Studieren Sie insbesondere die Definition der Struktur **`struct worm`** und die Übergabe der Strukturvariable `userworm` per Adresse an andere Funktionen.
Versuchen Sie auch, die Implementierung des Moduls `messages.c` zu verstehen, das Sie im Rahmen dieses Blatts bereits fertig implementiert zur Verfügung gestellt bekommen haben.
Rechnen Sie damit, dass Ihnen bei der Abnahme der nächsten Vorführaufgabe Fragen zum Code gestellt werden.

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`