

Praktikum zu „Grundlagen der Programmierung“

Blatt 5

Lernziele: Implementierung des Löschens am Wurmende durch Array-Datenstruktur; Entdeckung von Selbstkollisionen des Wurms
Erforderliche Kenntnisse: Arbeiten mit Arrays und Schleifen
Voraussetzungen:

- Vollständige Bearbeitung des letzten Blattes 04 (Worm010).

Übersicht

Im Rahmen dieses Aufgabenblatts implementieren wir die Version `Worm020`, die das Löschen am Ende des Wurms und die Entdeckung von Kollisionen des Wurms mit sich selbst zum wesentlichen Gegenstand hat. Um die Löschung implementieren zu können, müssen wir offensichtlich die Position (Zeile, Spalte) des letzten Elements des Wurms kennen.

An der Spitze kommt, aufgrund der Bewegung des Wurms, bei jedem Schritt eine neue Position für das Kopfelement hinzu. Am hinteren Ende können nach Löschung des letzten Elements die Koordinaten dieses letzten Elements vergessen werden. Das ehemals vorletzte Element ist im nächsten Schritt das neue letzte Element.

Falls der Wurm die Länge n hat und während der nächsten n Bewegungen des Wurms keine Kollisionen oder sonstige Fahrfehler auftreten, wird das derzeit aktuelle Kopfelement nach insgesamt n Schritten das letzte Element des Wurms sein und muss dann gelöscht werden. Seine Position muss also genau für n Schritte gespeichert werden.

Diese Überlegungen zeigen, dass wir eine Datenstruktur brauchen, in der wir gerade die n Positionen (Zeilen- und Spaltenkoordinaten) aller sichtbaren Wurmelemente speichern können. Ein paar zusätzliche Hilfsvariablen und ein geeignetes Management der Datenstruktur werden dafür sorgen, dass wir bei jedem Schritt wissen, wo in der Datenstruktur sich die Positionen der einzelnen Wurmelemente befinden, ohne dass nach jedem Schritt die Koordinaten innerhalb der Datenstruktur aufwendig „umkopiert“ werden müssen.

In einem ersten Schritt überlegen wir uns, wie diese Datenstruktur und deren Management genau funktionieren sollen. Erst danach implementieren wir unsere Idee im C-Programm!

Grundsätzliche Bemerkungen zur Organisation Ihres Codes

Eine ordentliche und übersichtliche Organisation von Entwicklungsdokumenten ist die Grundvoraussetzung für professionelle Software-Entwicklung. Zu den Entwicklungsdokumenten zählt unter anderem auch der Quellcode Ihrer Programme.

Für die Erteilung von Testaten für Ihre Implementierungen ist Grundvoraussetzung, dass Sie Ihren Code genau so organisieren, wie es in der Aufgabenstellung beschrieben wird. Dazu gehört insbesondere, dass Sie Verzeichnis- und Dateinamen, Schreibweise der Namen (groß/klein) und

Hierarchie der Verzeichnisse den Vorgaben entsprechend ausführen¹.


Eine Vorgabe wie `~/GdP1/Praktikum/Code/Worm020/worm.c` ist also durchaus ernst gemeint. Desweiteren müssen Sie Ihren Code ordentlich einrücken und übersichtlich gestalten. Ihr Editor hilft Ihnen dabei durch automatische Formatierung (`vim`: Tastenkombination `gg=G`). Sollte Ihre Implementierung diesen Vorgaben nicht genügen, müssen Sie damit rechnen, dass Ihr Betreuer Ihnen kein Testat ausstellt!

Überlegungen zur Modellierung des Wurms und zum Löschen am Wurmende

Im Folgenden überlegen wir uns, wie wir die Positionen der Wurmelemente speichern und die Bewegung des Wurms und das Löschen am Ende organisieren können.

Die linke der beiden nachfolgenden Tabellen zeigt den Inhalt eines kleinen Fensterpuffers mit 4 Zeilen (`LINES = 4`) und 5 Spalten (`COLS = 5`). Die Position in der linken oberen Ecke des Fensterpuffers hat bekanntlich die Koordinaten $(0, 0)$. Die Position in der rechten unteren Ecke hat die Koordinaten $(\text{LINES}-1, \text{COLS}-1)$. **Per Konvention der *curses*-Bibliothek nennen wir die Zeilenkoordinate zuerst.**

Die rechte Tabelle skizziert den Inhalt der Datenstruktur, in der wir die Positionen der einzelnen Wurmelemente speichern. In der Tabellenzeile, die ganz rechts die Bezeichnung Zeile trägt, merken wir uns die Zeilenkoordinaten. Entsprechend speichert die Tabellenzeile mit der Bezeichnung Spalte die Spaltenkoordinaten. Die spätere Implementierung in C wird zwei Arrays zur Speicherung der Zeilen- und Spaltenkoordinaten verwenden. Daher geben wir in der Tabellenzeile mit der Bezeichnung Index den für den Zugriff auf die beiden Koordinaten-Arrays benutzten Index an. In der letzten Tabellenzeile mit der Bezeichnung **H/T** skizzieren wir noch, in welchen Tabellenspalten gerade die Koordinaten des Kopfelements (*Head*) und die des letzten Elements (*Tail*) des Wurms gespeichert werden.

Die in den beiden Tabellen dargestellte Situation beschreibt den Fall, in dem gerade nur das Kopfelement  des Wurms an Position $(2, 0)$ dargestellt wird. Der Wurm erscheint also gerade erst im Fenster, und es ist noch kein Ende des Wurms zu sehen. Die übrigen Zellen der Datenstruktur sind noch unbenutzt, was wir mit einem **U** (für *unused*) kennzeichnen. Die rechte Tabelle hat 5 Spalten zur Speicherung der Koordinaten vorgesehen. Damit hat der Wurm im Beispiel eine maximale Länge von 5 Elementen.

Fensterpuffer				
0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4
3,0	3,1	3,2	3,3	3,4
4,0	4,1	4,2	4,3	4,4

Datenstruktur für Positionen					
0	1	2	3	4	Index
2	U	U	U	U	Zeile
0	U	U	U	U	Spalte
H					H/T

Nun wird der Wurm um eine Position bewegt. Im Beispiel soll er sich nach rechts bewegen. Es wird demnach ein neues Wurmelement an Position $(2, 1)$ gezeichnet. Das Element an Position $(2, 0)$, welches gerade noch das Kopfelement war, ist nun das zweite Element des Wurms. Da derzeit alle

¹Gilt auch dann, wenn das Betriebssystem nicht wirklich zwischen Groß- und Kleinschreibung unterscheiden kann!

Elemente des Wurms gleich aussehen, genügt es, nur das neue Kopfelement zu zeichnen.

In der Tabelle mit den Koordinaten der Wurmelemente fügen wir in der Spalte mit Index 1 die Koordinaten des neuen Kopfelementes ein. Zudem vermerken wir in der letzten Tabellenzeile, dass die Koordinaten des Kopfelementes sich in der Spalte mit Index 1 befinden.

Die Situation wird in den beiden folgenden Tabellen dargestellt.

Fensterpuffer				
0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4
3,0	3,1	3,2	3,3	3,4
4,0	4,1	4,2	4,3	4,4

Datenstruktur für Positionen					
0	1	2	3	4	Index
2	2	U	U	U	Zeile
0	1	U	U	U	Spalte
	H				H/T

Wir überspringen nun ein paar Schritte und zeigen die Situation, in der alle Spalten der rechten Tabelle mit Koordinaten gefüllt sind. Der Wurm hat dabei ein paar Haken geschlagen.

Die Koordinaten des Kopfelementes sind in der Spalte mit Index 4 abgelegt, der Wurm hat also seine maximale Länge von 5 Elementen erreicht. Der Wurm wird nun erstmals in seiner ganzen Länge dargestellt. Die Position $(2, 0)$ des letzten Elements ist in der Spalte mit Index 0 gespeichert. Wir skizzieren dies durch das T in der letzten Tabellenzeile.

Fensterpuffer				
0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4
3,0	3,1	3,2	3,3	3,4
4,0	4,1	4,2	4,3	4,4

Datenstruktur für Positionen					
0	1	2	3	4	Index
2	2	2	1	1	Zeile
0	1	2	2	3	Spalte
T				H	H/T

Der Wurm soll gerade wieder nach rechts kriechen, sein Kopf also im nächsten Schritt die Position $(1, 4)$ einnehmen. Die Durchführung des nächsten Schrittes zeigt nun sowohl die Idee hinter der Datenstruktur für die Wurmpositionen als auch die Bewältigung der Aufgabe des Löschens am Wurmende.

Natürlich wollen wir auch die neue Kopfposition $(1, 4)$ in der rechten Tabelle speichern, allerdings sind derzeit alle Spalten in unserer Datenstruktur belegt. Das ist jedoch kein Problem, denn der Wurm soll ja maximal nur 5 Elemente lang sein. Indem wir die Koordinaten des Wurmelements an der letzten Position $(2, 0)$ löschen, machen wir diesen Speicherplatz für die Speicherung des neuen Kopfelementes frei.

Allerdings verlieren wir durch die Löschung der Koordinaten der Schwanzposition die Information, wo wir im Fensterpuffer das Schwanzelement löschen müssen. Wir müssen also zuerst in einem *ersten Halbschritt* das derzeitige letzte Element des Wurms löschen. Danach können wir seine Koordinaten in der rechten Tabelle löschen und im frei werdenden Speicherplatz die Koordinaten des neuen Kopfelementes speichern.

Erster Halbschritt: (Löschen des letzten Elements im Fensterpuffer nebst seiner Koordinaten):

Fensterpuffer				
0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4
3,0	3,1	3,2	3,3	3,4
4,0	4,1	4,2	4,3	4,4

Datenstruktur für Positionen					
0	1	2	3	4	Index
	2	2	1	1	Zeile
	1	2	2	3	Spalte
T				H	H/T

Nun ist der Tabellenplatz zum Speichern der Koordinaten des neuen Kopfelements an Position (1, 4) frei.

Wir führen den *zweiten Halbschritt* durch, in dem wir das neue Kopfelement im Fensterpuffer speichern und seine Koordinaten in der rechten Tabelle in der Spalte mit Index 0 vermerken.

Die Koordinaten des neuen letzten Elements sind jetzt in der Spalte mit dem Index 1 gespeichert. Wir stellen die Situation in den beiden nachfolgenden Tabellen dar. Man beachte die letzte Zeile der rechten Tabelle mit den Einträgen **H** und **T** für die Koordinaten der aktuellen Kopf- und Schwanzenelemente.

Zweiter Halbschritt: (Darstellen des neuen Kopfelements und Speichern seiner Koordinaten):

Fensterpuffer				
0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4
3,0	3,1	3,2	3,3	3,4
4,0	4,1	4,2	4,3	4,4

Datenstruktur für Positionen					
0	1	2	3	4	Index
1	2	2	1	1	Zeile
4	1	2	2	3	Spalte
H	T				H/T

Im nächsten Schritt soll der Wurm nach oben an Position (0, 4) kriechen. Die erforderlichen beiden Halbschritte und die damit einhergehenden Veränderungen in der Datenstruktur sind in den nächsten 4 Tabellen dargestellt. Man beachte wieder die jeweils letzte Zeile in den rechten Tabellen.

Erster Halbschritt: (Löschen des letzten Elements im Fensterpuffer nebst seiner Koordinaten):

Fensterpuffer				
0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4
3,0	3,1	3,2	3,3	3,4
4,0	4,1	4,2	4,3	4,4

Datenstruktur für Positionen					
0	1	2	3	4	Index
1		2	1	1	Zeile
4		2	2	3	Spalte
H	T				H/T

Zweiter Halbschritt: (Darstellen des neuen Kopfelements und Speichern seiner Koordinaten):

Fensterpuffer				
0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4
3,0	3,1	3,2	3,3	3,4
4,0	4,1	4,2	4,3	4,4

Datenstruktur für Positionen					
0	1	2	3	4	Index
1	0	2	1	1	Zeile
4	4	2	2	3	Spalte
	H	T			H/T

Optimierung: Im zweiten Halbschritt werden die Koordinaten des neuen Kopfelements in die rechte Tabelle eingetragen. Wir können auf das explizite Löschen der Koordinaten im ersten Halbschritt verzichten.

Wenn wir nun alle bisher im Beispiel gezeigten Schritte Revue passieren lassen, erkennen wir folgende Eigenschaft unserer Datenstruktur, die für alle Schritte gleicher Massen gilt, also *invariant* ist:

Invariante: *Wenn das Ende des Wurms sichtbar ist, dann sind die Koordinaten des letzten Elements (Schwanzposition) genau in der Spalte der rechten Tabelle abgelegt, in der wir die nächste Position des Kopfelements speichern wollen.* Demzufolge brauchen wir uns nur den Spaltenindex für die aktuelle Kopfposition zu merken, in unserer Skizze also die mit **H** gekennzeichnete Spalte der rechten Tabelle. Die Spalte mit den Koordinaten für die Position des letzten Elements (Schwanzposition) erhalten wir dann durch Addition von 1. Natürlich müssen wir den so erhaltenen Index dann noch korrigieren, wenn wir das Ende der Tabelle überschreiten. Der korrekte Index für die Schwanzposition ist in diesem Fall dann die 0.

Mathematisch verbirgt sich hinter dieser Korrektur eine **Modulo-Operation**:

```
// Generell
IndexSchwanzPos = (IndexKopfPos + 1) modulo MaxWurmLänge
// Im Beispiel
IndexSchwanzPos = (IndexKopfPos + 1) modulo 5
```

In obiger Formulierung für die Invariante steht die Prämisse „Wenn das Ende des Wurms sichtbar ist ...“.

Ob das Ende des Wurms sichtbar ist, erkennen wir in unserer Datenstruktur daran, dass an der Indexposition für das Schwanzende noch ein **U** (für *unused*) eingetragen ist. Falls ja, ist das Ende des Wurms noch nicht sichtbar.

Bemerkung: Die hier in den jeweils rechten Tabellen skizzierte Datenstruktur nennt man Ringpuffer. Sie kommt insbesondere sehr häufig in der Systemprogrammierung (Betriebssysteme) vor, wo sie zum Beispiel zur Implementierung von Warteschlangen verwendet wird. Aus mehreren Gründen, etwa Erwägungen zu Performance, Vorhersagbarkeit und Robustheit, gibt man Arrays mit festen Obergrenzen, die als Ringpuffer verwaltet werden, oftmals den Vorzug vor sogenannten dynamischen Datenstrukturen (z.B. verketteten Listen), deren Speicherplatz nur nach Bedarf alloziert und bei Nichtbenutzung wieder de-alloziert wird.

Dies gilt insbesondere für Betriebssysteme mit harten Echtzeitanforderungen.

Nach diesen ausführlichen Betrachtungen zu einer Datenstruktur, die das Löschen am Ende des Wurms ermöglicht, wenden wir uns nun der Implementierung dieser Datenstruktur in der Programmiersprache C zu.

Aufgabe 1 (Kopieren der Version Worm010 nach Worm020)

Anstatt unsere Version Worm010 abzuändern, stellen wir eine Kopie in Worm020 her, damit wir auf der Kopie arbeiten können.

Wechseln Sie in einer Shell in das Verzeichnis ~/GdP1/Praktikum/Code:

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

Mit folgendem Befehl stellen wir eine Kopie des Verzeichnisses Worm010 in einem neuen Verzeichnis Worm020 her. Die Option `-r` steht für rekursives Kopieren.

```
$ cp -r Worm010 Worm020
```

Wechseln Sie in das neue Verzeichnis Worm020.

```
$ cd Worm020
```

Mit einem abschließenden `make clean` räumen wir alte Kompilate im Verzeichnis bin auf.

```
$ make clean
```

Ändern Sie nun noch die Datei `Readme.fabr` ab, indem Sie folgenden Inhalt eintragen:

```
New in this version
- Introduction of (separated) data structures for the worm.
We store length of the worm and its element positions on the screen.
- The worm has a visible end. We cleanup behind the worm.
- We detect self-collisions of the worm
```

Tip: Speichern Sie die neue Kopie gleich im Repository (`git add` und danach `git commit`)

Aufgabe 2 (Definition der globalen Datenstrukturen zur Modellierung des Wurms)

Wechseln Sie in das Verzeichnis `~/GdP1/Praktikum/Code/Worm020` und öffnen Sie die Datei `worm.c` in einem Editor.

Auf der Basis der vorherigen Überlegungen nehmen wir zunächst Änderungen in den Abschnitten für die C-Präprozessor-Definitionen und die globalen Variablen vor.

Abschnitt für Präprozessor-Definitionen: Im Absatz, der durch den Kommentar *Dimensions and Bounds* gekennzeichnet ist, fügen Sie die Definition der Konstanten `WORM_LENGTH` ein:

```
#define WORM_LENGTH 20    // Maximal length of the worm
```

Diese Konstante wird unter anderem für die Definition der Koordinaten-Arrays benötigt.

Abschnitt für globale Variablen: Bisher hatten wir die Position des Kopfelements in den beiden globalen Variablen `theworm_headpos_y` und `theworm_headpos_x` gespeichert.

Gemäß unserer Vorüberlegungen ersetzen wir diese beiden Variablen durch jeweils ein Array von Koordinaten mit jeweils `WORM_LENGTH` Elementen.

Ersetzen Sie also die Variablendefinitionen durch folgende Definitionen für zwei Arrays:

```
// Array of y positions for worm elements
int theworm_wormpos_y[WORM_LENGTH];
// Array of x positions for worm elements
int theworm_wormpos_x[WORM_LENGTH];
```

Wir haben oben bereits eine Präprozessor-Konstante `WORM_LENGTH` eingeführt, die wir in der gerade getätigten Definition der beiden Arrays auch zwingend benötigen. In C müssen die Größen von Arrays so angegeben werden, dass sie schon vom Compiler berechnet werden können.

Bei der Entwicklung und vor allem bei der Benutzung von Programmen zeigt sich manchmal,

dass man die maximale Größe von Arrays zwar in Form einer Präprozessor-Konstanten definiert, zur Laufzeit aber nur einen Teil des Arrays zur Benutzung freigeben möchte. Aus diesem Grund führt man häufig zusätzlich eine normale Integer-Variable ein, die zur Laufzeit die Obergrenze des benutzbaren Speicherplatzes im Array beschreibt. Durch Verwendung einer Variablen kann sich diese Obergrenze dann dynamisch während der Laufzeit ändern. Selbstverständlich ist darauf zu achten, dass die Variable nie Werte annimmt, die größer sind als die durch die Präprozessor-Konstante vorgegebene Grenze.

Wir folgen dieser Technik auch in unserem Programm und definieren zusätzlich zur Präprozessor-Konstanten `WORM_LENGTH` die globale Variable `theworm_maxindex`, die dynamisch den maximal zulässigen Index für den Zugriff auf die Arrays `theworm_wormpos_y` und `theworm_wormpos_x` festlegt.

```
// Last usable index into the arrays
// theworm_wormpos_y and theworm_wormpos_x
int theworm_maxindex;
```

Desweiteren folgt aus unseren Überlegungen, dass wir den Index speichern müssen, der die Position der Koordinaten des Kopfelements im Array beschreibt. Das ist die mit **H** gekennzeichnete Spalte in den in der Vorüberlegung rechts gezeigten Tabellen.

Wir speichern den Index in der globalen Variablen `theworm_headindex`:

```
// An index into the array for the worm's head position
// 0 <= theworm_headindex <= theworm_maxindex
int theworm_headindex;
```

Zusammen mit den bereits in der Version Worm010 eingeführten globalen Variablen `theworm_dx`, `theworm_dy` und `theworm_wcolor` haben wir nun alle Aspekte des Wurms durch mehrere globale Variablen modelliert.

Zusammenfassend sollte der Abschnitt für die globalen Variablen nun wie folgt aussehen:

```
//*****
// Global variables
//*****
// Data defining the worm
// They will become components of a structure, later
// Last usable index into the arrays
// theworm_wormpos_y and theworm_wormpos_x
int theworm_maxindex;
// An index into the array for the worm's head position
// 0 <= theworm_headindex <= theworm_maxindex
int theworm_headindex;
// Array of y positions for worm elements
int theworm_wormpos_y[WORM_LENGTH];
// Array of x positions for worm elements
int theworm_wormpos_x[WORM_LENGTH];
// The current heading of the worm
// These are offsets from the set {-1,0,+1}
int theworm_dx;
int theworm_dy;

enum ColorPairs theworm_wcolor; // Code of color pair used for the worm
```

Später, in der Version Worm050, werden wir alle diese globalen Variablen in einer einzigen Datenstruktur in Form eines Verbundes (`struct`) kapseln.

Aufgabe 3 (Änderungen an der Funktion `initializeWorm`) Nachdem wir die Datenstruktur des Wurms in Form mehrerer globaler Variablen modelliert haben, müssen wir nun an einigen schon bestehenden Funktionen Änderungen vornehmen, damit die in den Vorüberlegungen skizzierte Löschung am Ende des Wurms umgesetzt wird.

In Anbetracht Ihrer fortschreitenden Ausbildung ist der Anteil an vorgegebenem Code im Vergleich zu den vorherigen Aufgabenblättern deutlich geringer. Sie müssen also mehr Code selbst entwerfen.

Beginnen wir mit der Funktion `initializeWorm`. Die Funktion `initializeWorm` erhält einen zusätzlichen Parameter `len_max`, der die maximale Länge des Wurms bezeichnet. Die Signatur der Funktion `initializeWorm` lautet nun:

```
enum ResCodes initializeWorm(int len_max, int headpos_y,  
                             int headpos_x, enum WormHeading dir, enum ColorPairs color);
```

In der Funktion `doLevel` rufen wir die Funktion `initializeWorm` wie folgt auf:

```
res_code = initializeWorm(WORM_LENGTH, bottomLeft_y, bottomLeft_x , WORM_RIGHT,  
                          COLP_USER_WORM);
```

Ändern Sie, gemäß den obigen Vorgaben

- die Vorwärtsdeklaration der Funktion `initializeWorm` am Anfang der Datei.
- ihren Aufruf in der Funktion `doLevel`.
- die Kopfzeile der Definition der Funktion `initializeWorm`.

Als nächstes müssen Sie den Rumpf der Funktion `initializeWorm` umschreiben. Im Folgenden wird nur noch eine grobe Vorgabe gezeigt, die Ihnen den Weg weisen soll. Deklarationen für eventuell benötigte Hilfsvariablen für Schleifen etc. müssen Sie selbst einfügen.

In der Vorgabe wird die Präprozessor-Konstante `UNUSED_POS_ELEM` benutzt, die wir zum Markieren von unbenutzten Speicherstellen in den Koordinaten-Arrays benutzen wollen (vergl. Markierung **U** in den Vorüberlegungen). Fügen Sie folgende Definition im Abschnitt der Präprozessor-Konstanten hinzu:

```
// ### Codes for the array of positions ###  
// Unused element in the worm arrays of positions  
#define UNUSED_POS_ELEM -1
```

Hier nun die Vorgabe für den Rumpf der Funktion `initializeWorm`. Stellen, an denen Sie tätig werden müssen, sind wie üblich durch die Plathalter `@nnn` gekennzeichnet:

```
enum ResCodes initializeWorm(int len_max, int headpos_y,  
                             int headpos_x, enum WormHeading dir, enum ColorPairs color) {  
    // Local variables for loops etc.  
    @001  
  
    // Initialize last usable index to len_max -1  
    // theworm_maxindex  
    @002  
  
    // Initialize headindex  
    // theworm_headindex  
    @003
```



```

// Mark all elements as unused in the arrays of positions
// theworm_wormpos_y[] and theworm_wormpos_x[]
// An unused position in the array is marked
// with code UNUSED_POS_ELEM
@004

// Initialize position of worms head
theworm_wormpos_x[theworm_headindex] = @005
theworm_wormpos_y[theworm_headindex] = @005

// Initialize the heading of the worm
setWormHeading(dir);
// Initialize color of the worm
theworm_wcolor = color;
return RES_OK;
}

```

Aufgabe 4 (Änderungen an der Funktion **showWorm**)

In dieser Funktion müssen Sie im Aufruf von `placeItem` die Variablen `theworm_headpos_y` und `theworm_headpos_x` geeignet ersetzen.

Aufgabe 5 (Die neue Funktion **cleanWormTail**)

In den Vorüberlegungen zu diesem Blatt haben wir das Löschen am Ende des Wurms im sogenannten ersten Halbschritt erledigt. Die Durchführung dieses ersten Halbschritts implementieren wir nun in der neuen Funktion `cleanWormTail`. Das Speichern der neuen Kopfposition (zweiter Halbschritt) integrieren wir dagegen in einer späteren Teilaufgabe in die schon bestehende Funktion `moveWorm`.

Fügen Sie zunächst folgende Vorwärtsdeklaration für die neue Funktion `cleanWormTail` in den dafür vorgesehenen Abschnitt der Datei `worm.c` nach der Vorwärtsdeklaration der Funktion `showWorm` ein:

```
void cleanWormTail();
```

Die beiden erwähnten Halbschritte werden in unserer Implementierung in der Funktion `doLevel` aufgerufen. Fügen Sie direkt vor dem Aufruf der Funktion `moveWorm` den Aufruf der neuen Funktion `cleanWormTail` ein. Sie müssen nur den Aufruf genau so eintragen, wie nachstehend angegeben:

```

// Process userworm
// Clean the tail of the worm
cleanWormTail();
// Now move the worm for one step
moveWorm(&game_state);

```

Wie in den Vorüberlegungen dargestellt, müssen wir in der Funktion `cleanWormTail`, ausgehend vom Index des Kopfelements `theworm_headindex`, zunächst den Index des Schwanzelements errechnen. Das erreichen wir durch Inkrementieren und anschließender geeigneter Anwendung der Modulo-Funktion (%). Den so errechneten Index nutzen wir zum Auslesen der Koordinaten des Schwanzelements, wobei wir prüfen, ob die Schwanzposition im Array überhaupt schon benutzt ist (`UNUSED_POS_ELEM`). Falls sie schon benutzt ist, löschen wir das Wurmelement an diesen

Koordinaten im Bildschirmpuffer.

Für die konkrete Löschung des Schwanzelements wollen wir natürlich unsere Funktion `placeItem` verwenden. Diese verlangt jedoch als Argumente sowohl das zu platzierende Symbol als auch den Code für das zu verwendende Farbpaar (Vordergrund/Hintergrund). Wir kümmern uns zunächst um die Definition des Symbols und des Farbpaars.

Fügen Sie der Enumeration `ColorPairs` folgende zweite Komponente `COLP_FREE_CELL` hinzu, die für das Farbpaar einer leeren Zelle stehen soll:

```
enum ColorPairs {
    COLP_USER_WORM = 1,
    COLP_FREE_CELL,
};
```

Das zugehörige Symbol `SYMBOL_FREE_CELL` definieren wir gleich im Anschluss als Leerzeichen, ~‘wie folgt:

```
// Symbols to display
#define SYMBOL_FREE_CELL ' '
#define SYMBOL_WORM_INNER_ELEMENT '0'
```

Damit der oben definierte Code `COLP_FREE_CELL` für das Farbpaar verwendet werden kann, müssen wir das Farbpaar nun noch in der Funktion `initializeColors` initialisieren. Für die Farbgebung der freien Zelle wählen wir sowohl einen schwarzen Vordergrund als auch einen schwarzen Hintergrund. Demzufolge könnten wir jedes beliebige andere Symbol verwenden, da es durch die gewählte Farbgebung ohnehin nicht sichtbar wäre. Ändern Sie die Funktion `initializeColors` wie im Folgenden skizziert ab:

```
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);
}
```

Nun kommen wir zur Implementierung der Funktion `cleanWormTail`. Im Folgenden wird Ihnen eine Schablone vorgegeben, die Sie an den durch `@nnn` gekennzeichneten Stellen vervollständigen müssen.

Orientieren Sie sich dabei an den zu Beginn des Blatts gemachten Vorüberlegungen.

Fügen Sie die Implementierung von `cleanWormTail` hinter derjenigen der Funktion `showWorm` ein.

```
void cleanWormTail() {
    @006

    // Compute tailindex
    tailindex = @007

    // Check the array of worm elements.
    // Is the array element at tailindex already in use?
    // Checking either array theworm_wormpos_y
    // or theworm_wormpos_x is enough.
    if ( @008 ) {
        // YES: place a SYMBOL_FREE_CELL at the tail's position
        placeItem(@009, @009,
            SYMBOL_FREE_CELL, COLP_FREE_CELL);
    }
}
```

Hinweis: wie in den Vorüberlegungen bereits erwähnt, müssen wir nach dem Löschen des Schwanzelements in der Funktion `cleanWormTail` die Speicherposition der Koordinaten des Schwanzelements nicht extra löschen. Wir überschreiben diese Speicherstellen in der Funktion `moveWorm` einfach mit den Koordinaten des neuen Kopfelements.

Aufgabe 6 (Änderungen an der Funktion `moveWorm`)

Nun wenden wir uns den fälligen Änderungen an der Funktion `moveWorm` zu. Durch die Einführung einer aufwändigeren Datenstruktur für den Wurm sind ein paar Anpassungen in der Funktion notwendig geworden. Desweiteren können wir, aufgrund der reichhaltigeren Datenstruktur des Wurms, nun auch erstmals erkennen, ob der Wurm mit sich selbst kollidiert. In der Funktion `moveWorm` nutzen wir zur Kollisionserkennung die Hilfsfunktion `isInUseByWorm`, die wir dann in der letzten Teilaufgabe implementieren werden.

Vervollständigen Sie die nachfolgende Schablone an den durch @nnn gekennzeichneten Stellen:

```
void moveWorm(enum GameState* agame_state) {
    @010
    // Get the current position of the worm's head element and
    // compute the new head position according to current heading.
    // Do not store the new head position in the array of positions, yet.
    headpos_x = @011
    headpos_y = @012
    // 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 > getLastCol() ) {
        *agame_state = WORM_OUT_OF_BOUNDS;
    } else if (headpos_y < 0) {
        *agame_state = WORM_OUT_OF_BOUNDS;
    } else if (headpos_y > getLastRow() ) {
        *agame_state = WORM_OUT_OF_BOUNDS;
    } else {
        // We will stay within bounds.
        // Check if the worm's head will collide with itself at the new position
        if (isInUseByWorm(headpos_y, headpos_x)) {
            // That's bad: stop game
            *agame_state = WORM_CROSSING;
        }
    }

    // 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 theworm_headindex
        // Go round if end of worm is reached (ring buffer)
        @013
        // Store new coordinates of head element in worm structure
        theworm_wormpos_x[theworm_headindex] = @014
        theworm_wormpos_y[theworm_headindex] = @014
    }
}
```

Wie Sie der Schablone entnehmen können, wird zur Entdeckung von Kollisionen die Hilfsfunktion `isInUseByWorm` aufgerufen, der die neuen Koordinaten des Wurmkopfs übergeben werden. Falls

eine Kollision vorliegt, wird der neue Fehlercode `WORM_CROSSING` gesetzt. Fügen Sie diesen neuen Fehlercode der Enumeration `GameStates` wie folgt hinzu:

```
// Game state codes
enum GameStates {
    WORM_GAME_ONGOING,
    WORM_OUT_OF_BOUNDS,    // Left screen
    WORM_CROSSING,         // Worm head crossed another worm element
    WORM_GAME_QUIT,        // User likes to quit
};
```

Aufgabe 7 (Die neue Funktion `isInUseByWorm`)

In dieser Teilaufgabe implementieren wir die Funktion `isInUseByWorm`, die feststellt, ob der Wurm durch Vorrücken auf die neue Kopfposition, deren Koordinaten als Parameter übergeben werden, mit sich selbst kollidieren würde.

Fügen Sie zunächst, gemäß der folgenden Signaturvorgabe, eine Vorwärtsdeklaration der Funktion hinter derjenigen der Funktion `moveWorm` ein.

```
bool isInUseByWorm(int new_headpos_y, int new_headpos_x);
```

Die Funktion bekommt die potentiellen neuen Koordinaten des Kopfelements als Parameter übergeben. Bitte beachten Sie, dass zum Zeitpunkt des Aufrufs von `isInUseByWorm` diese potentiellen neuen Koordinaten des Kopfelements zwar schon durch `moveWorm` berechnet wurden, diese aber noch nicht in den globalen Datenstrukturen des Wurms gespeichert sind. Die Speicherung erfolgt nur, wenn keine Kollision oder andere Fehler vorliegen! Die Implementierung der Funktion `isInUseByWorm` müssen Sie im Wesentlichen selbst auf Basis der folgenden Vorgabe ausführen, in dem Sie an den durch @nnn gekennzeichneten Stellen ergänzen.

```
// A simple collision detection
bool isInUseByWorm(int new_headpos_y, int new_headpos_x) {
    int i;
    bool collision = false;
    i = theworm_headindex;
    do {
        // Compare the position of the current worm element with the new_headpos
        @015
    } while ( i != theworm_headindex &&
        theworm_wormpos_x[i] != UNUSED_POS_ELEM);
    // Return what we found out.
    return collision;
}
```

Dies war die letzte Teilaufgabe des Aufgabenblatts 5. Versuchen Sie nun, Ihr Programm zu kompilieren.

Desweiteren nicht vergessen, zumindest Ihre finale Version im Repository zu speichern!

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`