Technische Hochschule Ingolstadt
Fakultät für Informatik

Praktikum zu „Grundlagen der Programmierung“

Blatt 6 (Vorführaufgabe)

Lernziele: Techniken der getrennten Übersetzung
Erforderliche Kenntnisse: Inhalte aus der Vorlesung zum Thema der getrennten Übersetzung

Voraussetzungen:

- Vollständige Bearbeitung des letzten Blattes 05 (`Worm020`).

Übersicht

Dieses Aufgabenblatt widmet sich dem Thema der getrennten Übersetzung von Quelldateien (`*.c`) und dem abschließenden Binden (*Linking*) der Objektdateien (`*.o`) zu einem ausführbaren Programm.
Nur die Technik der getrennten Übersetzung und das Erzeugen von Bibliotheken macht die Entwicklung im Team, bisweilen sogar an verteilten Standorten und die Beherrschung von Programmen mit mehreren Millionen Zeilen C-Source-Code möglich.
Das Programm, welches wir im Rahmen des Praktikums erarbeiten, hat zwar in der Endversion `WormFinal` nur knapp 1500 Zeilen C-Code (ohne Leerzeilen), aber auch bei dieser noch überschaubaren Anzahl an Code-Zeilen lohnt sich der Einsatz der getrennten Übersetzung bereits[1].
Ausgehend von der Datei `worm.c` werden wir diese in mehrere Dateien aufteilen, wobei die Teile (Module[2]) nach inhaltlichen Gesichtspunkten zugeschnitten sind. Damit die Module einzeln übersetzt werden können, fügen wir für jedes Modul noch eine entsprechende Header-Datei (`*.h`) hinzu, welche die Deklarationen für Bezeichner enthält, die in anderen Modulen nutzbar sein sollen.
Wir bilden folgende Module:

- Dateien `board_model.h`, `board_model.c`:
  - Code für das Platzieren von Zeichen im Anzeigefenster
  - Auskunftsfunktion für Dimensionen des Anzeigefensters
  - (erst später:) Datenstruktur für das Spielbrett (wo sind Barrieren, Futterbrocken, ...)
- Dateien `prep.h`, `prep.c`:
  - Standard-Code für die Initialisierung des Anzeigefensters im Curses-Modus und seine Rücksetzung am Ende des Spiels
- Dateien `worm_model.h`, `worm_model.c`:

[1] Die bereinigte Anzahl (ohne Leerzeilen) der Zeilen C-Code in unserer aktuellen Version `Worm020` ermittelt sich im Verzeichnis `Worm020` wie folgt: `cat worm.c | sed -e '/^$/d' | wc`

[2] In diesem Kontext bedeutet Modul eine C-Datei, die einzeln übersetzt werden kann (Übersetzungseinheit)

  - Datenstrukturen des Wurms und deren Manipulation.
    Modul sollen alle Details der Implementierung des Wurms gekapselt sein

- Dateien `worm.h`, `worm.c`:
  - Das Hauptprogramm nebst Definitionen, die für alle Module gelten sollen
  - Anders ausgedrückt: der Rest, der nicht in andere Module verschoben wurde

Aufgabe 1 (Kopieren, Auspacken und Umbenennen des Templates)

Im Moodle-Kursraum finden Sie im Verzeichnis Praktikum die Datei `Worm030Template.zip`. Kopieren Sie diese Datei in das Verzeichnis `~/GdP1/Praktikum/Code`. In einer Shell wechseln Sie mit dem nachfolgenden Befehl in das Verzeichnis `~/GdP1/Praktikum/Code`.
`$ cd ~/GdP1/Praktikum/Code`
Vergewissern Sie sich mittels des Befehls `ls`, dass die Zip-Datei im Verzeichnis vorhanden ist.
`$ ls Worm030Template.zip`
Entpacken Sie die Zip-Datei durch den nachfolgenden Befehl.
Dadurch wird das Verzeichnis `Worm030Template` mit einigen Dateien darin angelegt.
`$ unzip Worm030Template.zip`
Listen Sie den Inhalt des neuen Unterverzeichnisses.
`$ ls -la Worm030Template`
Benennen Sie das Verzeichnis `Worm030Template` um in `Worm030`
`$ mv Worm030Template Worm030`
Listen Sie den Inhalt des umbenannten Unterverzeichnisses.
`$ ls -la Worm030`
Listen Sie den Inhalt des aktuellen Verzeichnisses.
`$ ls -la`
Löschen Sie das Zip-Archiv
`$ rm -f Worm030Template.zip`
Kopieren Sie die Datei `Worm020/worm.c` ins Verzeichnis `Worm030`.
Diese Datei werden wir im Folgenden in mehrere kleine Dateien aufteilen.
`$ cp Worm020/worm.c Worm030`
Fügen Sie das neue Verzeichnis `Worm030` mitteles `git add` und `git commit` Ihrem Repository hinzu.

Aufgabe 2 (Test der Ausgangssituation)

Öffnen Sie eine Shell und wechseln Sie in dieser Shell ins Verzeichnis `~/GdP1/Praktikum/Code/Worm030`. Eine Auflistung der Dateien in diesem Verzeichnis sollte folgenden Inhalt anzeigen:

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

Bis auf die Datei `worm.c`, die wir zum Schluss der letzten Aufgabe aus dem Verzeichnis der vorherigen Version `Worm020` kopiert haben, sind alle Dateien durch das Auspacken des Templates `Worm030Template.zip` entstanden.
Im Vergleich zur Vorversion `Worm020` hat sich das `Makefile` geringfügig geändert. Es enthält nun Regeln für die getrennte Übersetzung der einzelnen Modul-Dateien[3]. Inhalt und Zweck der Dateien `usage.txt` und `Readme.fabr` sind bekannt.
Die anderen Dateien aus dem Template sind bis auf Kommentare im Wesentlichen leer. Sie wurden aus Gründen einer strukturellen Vorgabe bereits angelegt und sollen in den folgenden Aufgaben von Ihnen mit Code gefüllt werden. Diese Code-Anteile sind derzeit noch alle in der Datei `worm.c` aus der Vorversion enthalten.
Da die neuen Dateien noch, bis auf Kommentare, leer sind und der komplette Code noch in `worm.c` vereint ist, sollte sich das gesamte Programm nach wie vor übersetzen lassen.
Überprüfen Sie diese Annahme durch Ausführung des folgenden Befehls:

```
$ 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 oben angegebenen Schritte korrekt ausgeführt.

Aufgabe 3 (Befüllen der Header-Datei `prep.h`)

Wir beginnen das Herauslösen der einzelnen Module, in dem wir für jedes Modul die bereits vorbereitete Header-Datei mit Deklarationen befüllen.
Wir beginnen mit der Header-Datei `prep.h`.
Im Folgenden werden Sie mehrmals Code-Passagen aus der Datei `worm.c` in eine andere Datei verlagern müssen. Hierzu benutzen Sie die Funktionen Ausschneiden, Kopieren und Einfügen Ihres Editors.
Falls Sie den Editor `vim` benutzen, zeigen Ihnen die folgenden Ausführungen ein paar dazu passende Tricks, die Sie natürlich im Manual des `vim` oder im `vimtutor` ausführlich nachlesen können.
Siehe hierzu auch:
https://www.ubuntupit.com/100-useful-vim-commands-that-youll-need-every-day/
Bei Benutzung eines anderen Editors, müssen Sie sich selbst informieren, wie Sie Textpassagen ausschneiden, kopieren und einfügen können.

Hinweis: Der Editor *Pluma* erlaubt das Un-Docking von Editor-Fenstern, d.h. einzelne Tabs (Reiter) können in eigenständige Frames herausgezogen werden.
Zurück zum Editor `vim`. Öffnen Sie die Datei `worm.c` mit dem Editor `vim`.

```
$ vim worm.c
```

Vergrößern Sie dann das Shell-Fenster so, dass es vertikal den ganzen Bildschirm einnimmt. Horizontal sollte das Fenster nur so breit sein, dass das Programm ohne Zeilenumbruch dargestellt werden kann (mehr ist Platzverschwendung).
Innerhalb der Editor-Session sehen Sie nun wahrscheinlich den Anfang der Datei `worm.c`, wobei am linken Rand die Zeilennummern der Datei ausgegeben werden. Falls Sie lieber mit der Maus markieren und nicht wie unten beschrieben Blöcke markieren, könnte die Anzeige der Zeilennummern stören. Daher schalten wir die Zeilennummerierung kurzzeitig ab.

[3] Eine Erläuterung der Datei `Makefile` und Erklärung des Werkzeugs `make` sind nicht Gegenstand dieses Blatts.

Wechseln Sie hierzu in die Befehlszeile des Editors. Durch Eingabe von ‚:‘gelangen Sie in die letzte Zeile des Editors. In dieser sogenannten Befehlszeile können Sie komplexe Befehle zur Textmanipulation eingeben. Geben Sie in der Befehlszeile den Befehl `set nonumber` ein. Die Zeilennummern am linken Rand sollten daraufhin verschwinden. Der Befehl `set number` schaltet sie bei Bedarf wieder ein.
Als nächstes teilen wir das Fenster des Editors in zwei Hälften (horizontal). Dazu wieder ‚:‘eingeben und dann in der Befehlszeile den Befehl `split` eingeben. Das Fenster teilt sich.
Die Tastenkombination Strg + w, Strg + w (also zweimal hintereinander Strg + w drücken) wechselt zwischen den Teilfenstern hin und her (ausprobieren).
Wir laden nun die Datei `prep.h` in den Editor und zeigen sie im unteren Fenster an. Wechseln Sie in das untere Fenster. Wechseln Sie mit ‚:‘in die Befehlszeile und geben Sie ein:
`e prep.h`
Der Buchstabe `e` steht hier für *edit*. Damit laden wir die Datei `prep.h` in den Editor und zeigen sie im aktuellen Teilfenster an. Bei der Eingabe des Dateinamens hilft Ihnen, wie in der Shell, die ⇄-Taste zur Vervollständigung. Die Pfeiltasten ←/→ erlauben das Editieren der Eingabe.
Nachdem Sie die Eingabe-Taste gedrückt haben, wird die Datei `prep.h` in das untere Fenster des Editors geladen. Sie sollten dort nun folgenden Inhalt sehen (Kommentare ausgelassen):

```
// ...
// Basic functions for initialization and cleanup of curses applications
#ifndef _PREP_H
#define _PREP_H

#endif  // #define _PREP_H
```

Die drei bereits in der Datei befindlichen `CPP`-Statements `#ifndef`, `#define` und `#endif` bilden das in der Vorlesung vorgestellte Muster, das die mehrfache Expansion von Dateien durch den Makro-Prozessor `CPP` verhindert (siehe C-Tutorium, Abschnitt zum C-Präprozessor). Nach dem `CPP`-Statement `#define` und vor dem `CPP`-Statement `#endif` werden Sie nun Teile aus der Datei `worm.c` einfügen.
Unser Ziel ist, dass die Datei `prep.h` die Vorwärtsdeklarationen der Funktionen

- `initializeCursesApplication` und
- `cleanupCursesApplication`

enthält, denn das Modul `prep.c` soll die Implementierungen dieser Funktionen kapseln.
Wechseln Sie in das obere Teilfenster, das die Datei `worm.c` enthält (Strg + w, Strg + w). Suchen Sie die Vorwärtsdeklaration der beiden oben genannten Funktionen (‚/‘wechselt in den Suchmodus).
Wir suchen den folgenden Textblock in der Datei `worm.c`, um ihn auszuschneiden:

```
// Standard curses initialization and cleanup
void initializeCursesApplication();
void cleanupCursesApp(void);
```

Markieren Sie die erste der obigen drei Zeilen als Anfang eines Blocks mit der Marke *a*. Sie können Marken von *a*-*z* verteilen, also insgesamt 26 Stück. Bewegen Sie dazu einfach den Cursor in die erste Zeile des zu bildenden Blocks und drücken Sie nacheinander die Tasten m und a.
Die Betätigung der Taste m steht hierbei für *mark*, und das a ist der Name der Marke.

Danach fahren Sie mit dem Cursor in die letzte Zeile des zu kopierenden Blocks und drücken die Taste ‚:', um in die Befehlszeile zu gelangen. Jetzt teilen Sie dem Editor mit, dass Sie alles ab der Marke *a* bis zur aktuellen Zeile (letzte Zeile des Blocks, in der gerade der Cursor steht) löschen möchten. Die gelöschte Passage befindet sich danach in einem Zwischenpuffer, vergleichbar der Zwischenablage von Windows. Der skizzierte Löschbefehl wird wie folgt in der Befehlszeile des Editors eingegeben: `'a,.d`
Das Apostroph `'` referenziert die Marke *a* als Anfangsposition, der `.` referenziert immer die aktuelle Cursorposition und *d* steht für *delete*. Damit weisen wir den Editor an, alles ab Marke *a* bis zur aktuellen Cursorposition zu löschen und in den Zwischenpuffer zu kopieren. Statt der Position `.` (aktuelle Cursorposition) könnten Sie natürlich auch eine andere zuvor definierte Marke, etwa *b* mittels `'b` referenzieren.
Als Ergebnis des Löschbefehls sollten die 3 Zeilen des markierten Blocks aus der Datei `worm.c` verschwinden. Die drei Zeilen befinden sich in einem Zwischenpuffer. Wechseln Sie in das untere Fenster (Strg + w, Strg + w) und positionieren Sie den Cursor in der Zeile nach dem CPP-Statement `#define`.
Drücken Sie nun die Taste p für *paste* (Einfügen). Die drei Zeilen aus dem Zwischenpuffer sollten jetzt in die Datei `prep.h` eingefügt werden, deren Inhalt nun etwa so aussieht (Ohne Kommentare am Anfang):

```
// Basic functions for initialization and cleanup of curses applications

#ifndef _PREP_H
#define _PREP_H

// Standard curses initialization and cleanup
void initializeCursesApplication();
void cleanupCursesApp(void);

#endif  // #define _PREP_H
```

Hinweis: Wenn Sie oben beim Löschen statt der Befehlssequenz `'a,.d` die Sequenz `'a,.y` eingeben, dann wird der Block nicht gelöscht (*cut*), sondern nur kopiert (*copy*). Die Taste y steht hier für *yank* („to yank something", deutsch: etwas herausziehen).
Zum Abschluss stellen Sie den beiden Funktions-Deklarationen noch jeweils das Schlüsselwort `extern` voran, um die Natur dieser Deklarationen zu verdeutlichen.
Wie in der Vorlesung besprochen, ist das Voranstellen des Schlüsselwortes `extern` bei der Deklaration von Funktionen optional, da der Compiler bei Funktionen ohnehin automatisch die Speicherklasse `extern` annimmt. Soll eine globale Variable lediglich deklariert aber nicht definiert werden, so ist das Voranstellen von `extern` Pflicht, da sonst die Variable bereits definiert wird.
Speichern Sie nun die neue Version der Datei `prep.h` und die veränderte Version der Datei `worm.c`.
Wechseln Sie dazu ins jeweilige Fenster des Editors und geben Sie den Befehl `:w` für speichern ein. Hiermit ist die Erstellung der Header-Datei `prep.h` beendet. Wir haben also lediglich drei Zeilen aus der Datei `worm.h` in die Datei `prep.h` verschoben.
Die Beschreibung dieser einfachen Aktion war aber etwas länger, da wir das Auschneiden (`d` = *delete*) bzw. Kopieren (`y` = *yank*) und das anschließende Einfügen von Textblöcken (`p` = *paste*) in verschiedenen Fenstern (Puffern) des Editors vim genau beschrieben haben.
Für die Beschreibung der folgenden Aktionen gehen wir davon aus, dass Sie diese Technik nun beherrschen.

Aufgabe 4 (Befüllen der Header-Datei `board_model.h`)

Als nächstes lösen wir die Deklarationen der Funktionen

```
void placeItem(int y, int x, chtype symbol, enum ColorPairs color_pair);
int getLastRow();
int getLastCol();
```

aus der Datei `worm.c` heraus und verschieben sie in die dafür vorgesehene Header-Datei `board_model.h`. Das Modul `board_model.c` soll alle Funktionen kapseln, die etwas mit der Darstellung von Gegenständen auf dem Spielbrett zu tun haben. Die Bildung eines eigenen Moduls wird erst in späteren Versionen des Programms klarer motiviert. Dann nämlich, wenn wir eine spezielle Datenstruktur für die Speicherung der sich auf dem Spielbrett befindenden Gegenstände (Hindernisse, Futterbrocken, andere Würmer) einführen.
Falls Sie noch Ihre `vim`-Sitzung von der vorherigen Aufgabe geöffnet haben (die mit den beiden Fenstern), können Sie diese weiterhin verwenden. Ansonsten stellen Sie die Situation wieder her.
Wechseln Sie in das untere der beiden Editor-Fenster und laden Sie die Header-Datei `board_model.h` mittels Befehl
`:e board_model.h`
Markieren Sie die oben genannten drei Funktions-Deklarationen als Block (Anfang mittels Marke *a*, Ende des Block durch Positionierung des Cursors auf die letzte Zeile des Blocks) und löschen Sie den Block aus `worm.c` mittels Befehl
`:'a,.d`
Fügen Sie die gelöschten Zeilen mittels `p` in die Datei `board_models.h` ein (direkt vor `#endif`) und stellen Sie wieder bei allen drei Deklarationen das Schlüsselwort `extern` voran.
Beim Betrachten der Deklaration von `placeItem` fällt auf, dass in der Signatur der Funktion die Datentypen `chtype` und `enum ColorPairs` verwendet werden. Der Datentyp `chtype` wird in der Header-Datei `curses.h` der Curses-Bibliothek definiert, der Datentyp `enum ColorPairs` hingegen bisher in der Datei `worm.c` Damit die Deklaration von `placeItem` korrekt durchgeführt werden kann, müssen diese Datentypen zum Zeitpunkt der Deklaration dem Compiler bekannt sein. Das war bisher durch die Reihenfolge der Deklarationen und Definitionen in der Datei `worm.c` gegeben.
Damit die Deklaration von `placeItem` auch in der neuen Header-Datei `board_model.h` funktioniert, müssen wir entsprechende Definitionen dieser Datentypen vorher laden. Wir bewerkstelligen dies durch die folgenden beiden `#include`-Statements, die wir noch vor den Funktions-Deklarationen in `board_model.h` positionieren.

```
#include <curses.h>
#include "worm.h"
```

Die Datei `worm.h` wird erst in einer späteren Teilaufgabe erzeugt. Sie wird die Definition des Datentyps **`enum`** `ColorPairs` enthalten.
Das `CPP`-Statement
`#include <curses.h>`
bewirkt, dass die Datei `curses.h` in einer vorkonfigurierten Menge von Systempfaden gesucht wird. Das `CPP`-Statement `#include "worm.h"` bewirkt, dass die Datei `worm.h` im aktuellen Verzeichnis gesucht wird.
Ihre Datei `board_model.h` sollte nun wie folgt aussehen (Kommentare gekürzt):

```
// The board model

#ifndef _BOARD_MODEL_H
#define _BOARD_MODEL_H

#include <curses.h>
#include "worm.h"

extern void placeItem(int y, int x, chtype symbol, enum ColorPairs color_pair);

// Getters
extern int getLastRow();
extern int getLastCol();

#endif  // #define _BOARD_MODEL_H
```

Speichern Sie sowohl die neue Version der Datei `worm.c` als auch die Datei `board_model.h`

Aufgabe 5 (Befüllen der Header-Datei `worm_model.h`)

Nun lösen wir die Definition des Datentyps **enum** `WormHeading` und die Deklarationen der Funktionen

```
enum ResCodes initializeWorm(int len_max, . . . enum ColorPairs color);
void showWorm();
void cleanWormTail();
void moveWorm(enum GameStates* agame_state);
bool isInUseByWorm(int new_headpos_y, int new_headpos_x);
void setWormHeading(enum WormHeading dir);
```

aus der Datei `worm.c` heraus und verschieben sie in die dafür vorgesehene Header-Datei `worm_model.h`.
Das Modul `worm_model.c` soll alle Datentypen und Funktionen kapseln, die etwas mit der Speicherung und Manipulation/Bewegung des Wurms zu tun haben. In der Header-Datei `worm_model.h` deklarieren wir nur die Zugriffsfunktionen, die in anderen Modulen benutzt werden sollen. Dazu später mehr, wenn wir die Datei `worm_model.c` befüllen.
Falls Sie noch Ihre `vim`-Sitzung von der vorherigen Aufgabe geöffnet haben (die mit den beiden Fenstern), können Sie diese weiterhin verwenden. Ansonsten stellen Sie die Situation wieder her.
Wechseln Sie in das untere der beiden Editor-Fenster und laden Sie die Header-Datei `worm_model.h` mittels Befehl
`:e worm_model.h`
Markieren Sie in einem ersten Schritt die oben genannte Datentyp-Deklaration für **enum** `WormHeading` und verschieben Sie sie in die Datei `worm_model.h`. Danach verschieben Sie die Funktions-Deklarationen in die gleiche Header-Datei und stellen bei den Funktions-Deklarationen wieder das Schlüsselwort **extern** voran.
Beim Betrachten der in den verschobenen Deklarationen vorkommenden Datentypen sehen wir, dass wir den Datentyp **bool** sowie diverse Aufzählungstypen nutzen, die bisher in `worm.c` definiert sind.
Wie vorhin bei `board_model.h` kompensieren wir dies durch Voranstellen der folgenden beiden `CPP`-Statements:

```
#include <stdbool.h>
#include "worm.h"
```

Ihre Datei `worm_model.h` sollte nun wie folgt aussehen (Kommentare gekürzt, umformatiert):

```
// The worm model
#ifndef _WORM_MODEL_H
#define _WORM_MODEL_H

#include <stdbool.h>
#include "worm.h"

enum WormHeading {
WORM_UP,
WORM_DOWN,
WORM_LEFT,
WORM_RIGHT
};

extern enum ResCodes initializeWorm(int len_max, int headpos_y,
int headpos_x, enum WormHeading dir, enum ColorPairs color);
extern void showWorm();
extern void cleanWormTail();
extern void moveWorm(enum GameStates* agame_state);
extern bool isInUseByWorm(int new_headpos_y, int new_headpos_x);

//Setters
extern void setWormHeading(enum WormHeading dir);
#endif  // #define _WORM_MODEL_H
```

Speichern Sie sowohl die neue Version der Datei `worm.c` als auch die Datei `worm_model.h`.

Aufgabe 6 (Befüllen der Header-Datei `worm.h`)

Die letzte der Header-Dateien `worm.h` soll alle bisher in `worm.c` enthaltenen Definitionen für Aufzählungstypen (**`enum`**) und CPP-Konstanten (*`#define`*) enthalten. Damit kann diese Datei, wie oben bereits in `board_model.h` und `worm_model.h` geschehen, in anderen Modulen inkludiert werden.
Falls Sie noch Ihre `vim`-Sitzung von der vorherigen Aufgabe geöffnet haben (die mit den beiden Fenstern), können Sie diese weiterhin verwenden. Ansonsten stellen Sie die Situation wieder her.
Wechseln Sie in das untere der beiden Editor-Fenster und laden Sie die Header-Datei `worm.h` mittels Befehl

```
:e worm.h
```

Markieren Sie alle Zeilen ab Definition des Typs **`enum`** `ResCodes` bis einschließlich Definition des Typs **`enum`** `GameStates` und verschieben Sie alles in die Header-Datei `worm.h`.
Da keine besonderen Datentypen referenziert werden, brauchen wir diesmal keine zusätzlichen *`#include`*-Statements, um fehlende Definitionen zu laden.
Nach dem Verschieben sollte Ihre Datei `worm.h` wie folgt aussehen (Kommentare gekürzt):

```
#ifndef _WORM_H
#define _WORM_H

// Result codes of functions
enum ResCodes {
RES_OK,
RES_FAILED,
};


// Dimensions and bounds
#define NAP_TIME    100   // Time in milliseconds to . . .
```

```
#define MIN_NUMBER_OF_ROWS  3  // The guaranteed number of . . .
#define MIN_NUMBER_OF_COLS 10  // The guaranteed number . . .
#define WORM_LENGTH 20  // Max length of a worm
// Codes for the array of positions
#define UNUSED_POS_ELEM -1  // Unused element in the positions arrays

// Numbers for color pairs used by curses macro COLOR_PAIR
enum ColorPairs {
COLP_USER_WORM = 1,
COLP_FREE_CELL,
};

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

// 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
};

#endif  // #define _WORM_H
```

Speichern Sie sowohl die neue Version der Datei `worm.c` als auch die Datei `worm.h`.

Aufgabe 7 (Korrektur der Datei `worm.c` und Zwischentest der Übersetzung)

In den vorherigen Aufgaben haben wir Deklarationen von Funktionen und Definitionen von Datentypen aus der Datei `worm.c` gelöscht und in verschiedene neue Header-Dateien verschoben. Im jetzigen Zustand ist die Datei `worm.c` nicht mehr übersetzbar, da ja gewisse Teile ersatzlos gelöscht wurden.
Wir können aber in der Datei `worm.c` die neuen Header-Dateien gleich zu Beginn per `#include` einlesen, so die fehlenden Deklarationen und Definitionen laden, und sie damit dem Compiler wieder bekannt geben. Dadurch erhalten wir wieder ein übersetzbares Projekt.
Fügen Sie folgende `#include`-Statements in der Datei `worm.c` gleich hinter dem bisher letzten CPP-Statement `#include <unistd.h>` ein:

```
#include "prep.h"
#include "worm.h"
#include "worm_model.h"
#include "board_model.h"
```

Hinweis: Die Reihenfolge der Anweisungen ist beliebig.
Prüfen Sie sodann, ob Ihr Projekt wieder übersetzbar ist:
`$ 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 oben angegebenen Schritte korrekt ausgeführt.

Aufgabe 8 (Befüllen der Moduldatei `prep.c`)

Die meiste Arbeit ist nun schon getan. Wir müssen nur noch die einzelnen Teile der Implementierung aus der Datei `worm.c` in die entsprechenden Moduldateien verschieben. Die aufwändigere Erstellung der Header-Dateien liegt schon hinter uns.
Beginnen wir dazu wieder mit dem Modul `prep.c`, weil es die geringste Komplexität hat.
Falls Sie noch Ihre `vim`-Sitzung von der vorherigen Aufgabe geöffnet haben (die mit den beiden Fenstern), können Sie diese weiterhin verwenden. Ansonsten stellen Sie die Situation wieder her.
Im oberen Fenster sollte die aktuelle Version der Datei `worm.c` geladen sein. Wechseln Sie in das untere der beiden Editor-Fenster und laden Sie dort die bisher leere Moduldatei `prep.c` mittels Befehl
`:e prep.c`
Verschieben Sie nun den Code für die Implementierung der Funktionen

```
void initializeCursesApplication()
void cleanupCursesApp(void)
```

aus der Datei `worm.c` (hier löschen) in die neue Moduldatei `prep.c` (da einfügen). Im Code dieser Funktionen nutzen wir einige Funktionen der Curses-Bibliothek. Außerdem möchten wir die Deklarationen der Header-Datei `prep.h` als Vorwärtsdeklarationen im Modul `prep.c` nutzen.
Aus diesem Grund fügen Sie noch ganz am Anfang der Datei `prep.c`, gleich direkt nach den Kommentaren am Dateianfang, die folgenden beiden `CPP`-Statements ein:

```
#include <curses.h>
#include "prep.h"
```

Damit sollte das Projekt `Worm030` wieder in einem übersetzbaren Zustand sein, denn wir haben nur Code verlagert. Testen Sie diese Annahme durch 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, dann wissen Sie, was jetzt kommt:
→ Sie haben nicht alle oben angegebenen Schritte korrekt ausgeführt!

Aufgabe 9 (Befüllen der Moduldatei `board_model.c`)

Nun verschieben wir den Code für die Implementierung der Funktionen

```
void placeItem(int y, int x, chtype symbol, enum ColorPairs color_pair)
int getLastRow()
int getLastCol()
```

aus der Datei `worm.c` (hier löschen) in die neue Moduldatei `board_model.c` (da einfügen).
Ähnlich wie im Modul `prep.c` nutzen wir in den verschobenen Funktionen Definitionen und Funktionen der Curses-Bibliothek sowie Definitionen und Deklaration, die mittlerweile in der Header-Datei `worm.h` enthalten sind. Außerdem möchten wir die Deklarationen der Header-Datei `board_model.h` als Vorwärtsdeklarationen im Modul `board_model.c` nutzen.
Aus diesem Grund fügen Sie noch ganz am Anfang der Datei `board_model.c`, gleich direkt nach den Kommentaren am Dateianfang, die folgenden drei `CPP`-Statements ein:

```
#include <curses.h>
#include "worm.h"
#include "board_model.h"
```

Damit sollte das Projekt `Worm030` wieder in einem übersetzbaren Zustand sein, denn wir haben nur Code verlagert. Testen Sie diese Annahme wieder durch 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, ...

Aufgabe 10 (Befüllen der Moduldatei `worm_model.c`)

Als letztes verschieben wir noch die Definitionen der globalen Variablen für die Datenstruktur des Wurms

```
int theworm_maxindex;
int theworm_headindex;
int theworm_wormpos_x[WORM_LENGTH];
int theworm_wormpos_y[WORM_LENGTH];
int theworm_dx;
int theworm_dy;
enum ColorPairs theworm_wcolor;
und den Code für folgende Funktionen
extern enum ResCodes initializeWorm( . . . );
extern void showWorm();
extern void cleanWormTail();
extern void moveWorm(enum GameStates* agame_state);
extern bool isInUseByWorm(int new_headpos_y, int new_headpos_x);
extern void setWormHeading(enum WormHeading dir);
```

in die Moduldatei `worm_model.c`. Damit sind alle Datenstrukturen und Funktionen, die etwas mit der konkreten Modellierung und Manipulation des Wurms zu tun haben, in ein Modul ausgelagert und darin gekapselt.
In der zum Modul gehörenden Header-Dateien haben wir jedoch nur die obigen Funktionen deklariert und den Datentyp **`enum`** `WormHeading` definiert. Die Definition der globalen Variablen ist für die anderen Module nicht interessant, da diese, wenn überhaupt, über die Funktionen zugreifen. Die Interna der Datenstruktur, etwa die Tatsache, dass wir bisher globale Variable zur Modellierung nutzen, soll verborgen bleiben[4].
Das verschafft uns die Freiheit, in späteren Versionen die Datenstruktur des Wurms zu ändern, ohne dass der Code in den anderen Modulen angepasst werden muss.
Nachdem Sie, wie oben skizziert, die Definition der globalen Variablen für die Datenstruktur des Wurms und die genannten Funktionen aus der Datei `worm.c` in die Moduldatei `worm_model.c` verschoben haben, müssen Sie, analog zu den vorherigen Verschiebungen, noch ein paar `#include`-Statements einfügen.
Der verschobene Code nutzt Definitionen und Funktionen der Curses-Bibliothek sowie Definitionen und Deklaration, die mittlerweile in den Header-Dateien `worm.h` und `board_model.h` enthalten sind. Außerdem möchten wir die Deklarationen der Header-Datei `worm_model.h` als Vorwärtsdeklarationen im Modul `worm_model.c` nutzen.
Aus diesem Grund fügen Sie noch ganz am Anfang der Datei `worm_model.c`, gleich direkt nach

[4] Um die Nutzung der globalen Variablen eines Moduls durch andere Module tatsächlich unmöglich zu machen, müssten wir die globalen Variablen als **`static`** deklarieren. Diese Technik lernen wir aber erst später kennen.

den Kommentaren am Dateianfang, die folgenden CPP-Statements ein:

```
#include <curses.h>
#include "worm.h"
#include "board_model.h"
#include "worm_model.h"
```

Damit sollte das Projekt Worm030 wieder in einem übersetzbaren Zustand sein, denn wir haben nur Code verlagert. Testen Sie diese Annahme wieder durch 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, ...

Endkontrolle

Nachdem wir große Teile des Codes aus worm.c in andere Module verschoben haben, verbleiben nur noch wenige Funktionen in der Datei worm.c. Um Ihnen die Kontrolle Ihrer Umbauten zu erleichtern, sind im Folgenden die noch in der Datei worm.c verbliebenen Bestandteile skizziert. Funktionsrümpfe und Kommentare sind gekürzt:

```
#include <curses.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <string.h>
#include <unistd.h>

#include "prep.h"
#include "worm.h"
#include "worm_model.h"
#include "board_model.h"

// *********************************
// Management of the game
// *********************************
void initializeColors() { . . . }
void readUserInput(enum GameStates* agame_state ) { . . . }
enum ResCodes doLevel() { . . . }

//*****************************************************************************
// MAIN
//*****************************************************************************
int main(void) { ... }
```

Schlußbemerkung

Im Rahmen dieses Aufgabenblatts haben wir die vormals monolithische Implementierung der Datei Worm020/worm.c in mehrere nach Inhalten geordnete Module aufgespalten. Damit haben wir die Grundlage für die in späteren Versionen noch durchzuführenden Erweiterungen unseres Projekts gelegt und einen großen Fortschritt in Bezug auf die Wartbarkeit und Flexibilität unseres Codes erzielt.
Es ist von großer Wichtigkeit, dass Sie alle durchgeführten Teilschritte verstehen. Studieren Sie ins-

besondere die Rolle der Header-Dateien, die einen modularen Aufbau und die getrennte Übersetzung von Modulen erst praktikabel machen.

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`