Programmieren in C

Prof. Dr. Peter Thiemann
Hannes Saffrich
Sommersemester 2021

Universität Freiburg
Institut für Informatik

Übungsblatt 11

Abgabe: Montag, 12.07.2021, 9:00 Uhr morgens

In diesem Übungsblatt geht es um Funktionszeiger.

Funktionen sind das wichtigste Werkzeug zur Abstraktion: sie erlauben uns ein Stück Code zu schreiben und diesen Code auf verschiedenen Daten auszuführen.

Mit Funktionszeigern können wir Funktionen wie normale Werte behandeln – also die Adresse einer Funktion einer Variablen zuweisen, diese durch unser Programm reichen, und bei Bedarf die Funktion, deren Adresse in der Variable steht, aufrufen.

Insbesondere können wir sogenannte *Funktionen höherer Ordnung (higher-order functions)* schreiben, die einen Zeiger auf eine andere Funktion als Argument nehmen. Diese Technik ist sehr mächtig, da solche Funktionen nicht immer starr den selben Code auf unterschiedlichen Daten ausführen, sondern einen Teil des auszuführenden Codes selbst als Argument übergeben bekommen.

Als Beispiel wurde hierzu in der Vorlesung die Funktion `vec_iter` gezeigt, die einen Vektor und einen Funktionszeiger als Argumente nimmt und die übergebene Funktion auf jedem Element des Vektors aufruft.

Verwendet man eine Library (wie `Vec`) in einer Anwendung (wie dem Asteroiden-Spiel), so kann man auf diese Weise in der Anwendung Library-Code aufrufen, der wiederum Anwendungscode ausführt, ohne dass man den Library-Code editieren muss. Zum Beispiel könnte man die `vec_iter`-Funktion mit dem Asteroiden-`Vec` und einer Funktion aufrufen, die die Position eines Asteroiden verändert. Die `Vec`-Library kann so unabhängig von konkreten Anwendungen bleiben und deshalb auch in anderen Anwendungen verwendet werden.

C unsterstützt im Gegensatz zu Python, Rust und C++ leider keine anonymen Funktionen (Lambdas) und Funktionen, die lokale Variablen aus ihrer Umgebung einfangen können (Closures). Diese Features machen viele Anwendungen von Funktionen höherer Ordnung deutlich ergonomischer.

Aufgabe 11.1 (Higher-Order Functions; 4 Punkte)
In Ihrem Repository finden Sie in `blatt11` die folgenden Dateien:

- Die Dateien `ivec.h/.c` definieren den Datentyp `IntVec` – eine Variante des `Vec` aus den vorherigen Übungsblättern, der Elemente vom Typ `int` statt `void*` enthält.
- Die Dateien `ivec_ext.c` und `ivec_ext_test.c` sollen in dieser Aufgabe vervollständigt werden.

(a) (2 Punkte) Implementieren Sie die Funktion

```
void ivec_filter(IntVec* xs, bool (*f)(int));
```

die alle Elemente `x` aus `xs` entfernt, für die `f(x)` den Wert `false` zurückgibt.

Schreiben Sie mindestens einen Unittest für `int_vec_filter`.

(b) (2 Punkte) Implementieren Sie die Funktion

```
int ivec_fold(IntVec* xs, int y, int (*f)(int, int));
```

die alle Elemente von `xs` mit `f` kombiniert. Enthält `xs` zum Beispiel die Zahlen 1,2,3 und 4, so soll sich `ivec_fold(xs, y, f)` verhalten wie

```
f(f(f(f(y, 1), 2), 3), 4)
```

Ist `xs` leer so wird `f` gar nicht angewendet und direkt `y` zurückgegeben.

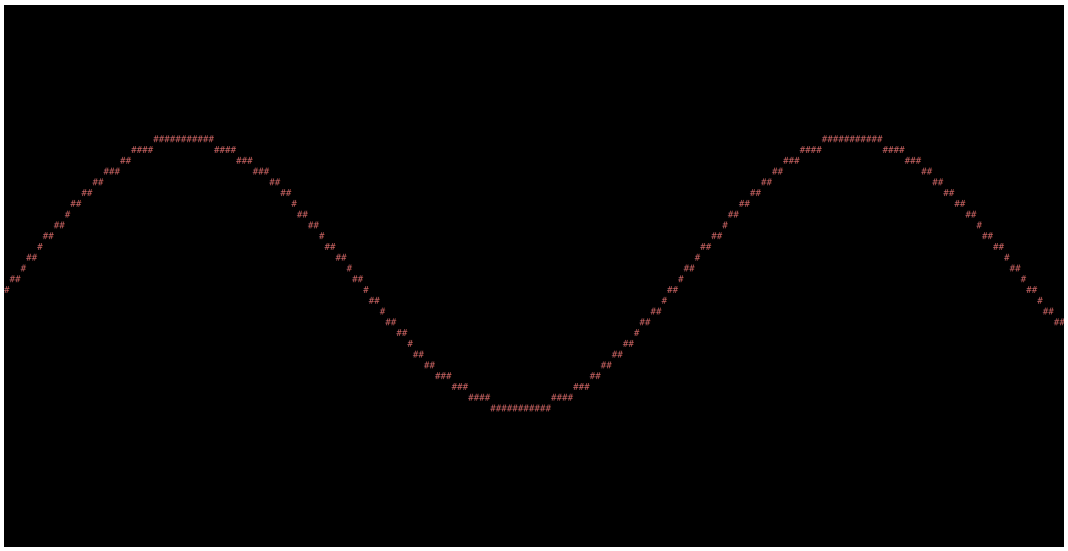
Verwenden Sie `ivec_fold` um die Funktionen

```
int ivec_sum(IntVec* xs);
int ivec_product(IntVec* xs);
```

zu implementieren, die die Summe bzw. das Produkt über alle Zahlen aus `xs` zurückgeben.

Schreiben Sie für `ivec_sum` und `ivec_product` jeweils mindestens einen Unittest.

Aufgabe 11.2 (Function Plotter; 10 Punkte)
In dieser Aufgabe sollen Sie mit der `tui`-Library einen Funktionsplotter schreiben.



Die API ist hierzu bereits in der `plot_lib.h` vorgegeben und soll in der `plot_lib.c` vervollständigt werden.

Die `plot_lib.h` definiert zwei Strukturen `Settings` und `Function` und deklariert eine einzige Funktion

```
void plot(Settings* settings, Function* functions, size_t num_functions);
```

Wird `plot` aufgerufen, so soll eine Terminal UI gestartet werden, die die `functions` mit den gegebenen `settings` zeichnet und durch Drücken der Taste `'q'` wieder beendet werden kann.

Die `Settings` geben an welcher Ausschnitt des Koordinatensystems der zu zeichnenden Funktionen im Terminal dargestellt werden soll:

```
typedef struct Settings {
  double x_min;  double x_max;
  double y_min;  double y_max;
} Settings;
```

Die Terminalkoordinate oben-links entspricht also der Funktionskoordinate `(x_min, y_min)` und die Terminalkoordinate unten-rechts entspricht der Funktionskoordinate `(x_max, y_max)`.

Eine `Function` enthält einen Zeiger auf die zu zeichnende Funktion und die Text- und Hintergrund-Farbe mit der die Funktion gezeichnet werden soll:

```
typedef struct Function {
  double (*f)(double, double);
  char* fg_color;
  char* bg_color;
} Function;
```

Um den Funktionsplotter künstlerisch interessanter zu machen, bilden die zu zeichnenden Funktionen nicht nur einen x-Wert auf einen y-Wert ab, sondern nehmen zusätzlich die Zeit seid Programmstart (in Sekunden) als Argument.[1]

Dies erlaubt es Animationen zu zeichnen, wie in folgendem Demo-Video zu sehen ist:

https://youtu.be/YASfZLJVEjM

Zum Beispiel kann eine Sinus-Funktion, die sich mit fortschreitender Zeit nach rechts bewegt, wie folgt definiert werden:

```
#include <math.h>

double moving_sin(double x, double t) {
  return sin(x - t);
}
```

Implementieren Sie die Funktion `plot` in der `plot_lib.c` und rufen Sie diese in der `main`-Funktion in `plot.c` mit mindestens zwei interessanten Funktionen ihrer Wahl auf.

Dabei sollen folgende Keybindings unterstützt werden:

- Die Taste `'q'` beendet die Terminal UI.
- Die Tasten `'+'` und `'-'` zoomen in und aus dem durch `settings` festgelegten Ausschnitt des Funktionskoordinatensystems. Dabei wird bei jedem Tastendruck der Ausschnitt um 20% größer bzw. kleiner.

Aufgabe 11.3 (Erfahrungen; 2 Punkte)

Notieren Sie Ihre Erfahrungen mit diesem Übungsblatt in der Datei `erfahrungen.txt` (benötigter Zeitaufwand, Probleme, Bezug zur Vorlesung, Interessantes, etc.).

Der Zeitaufwand *muss* dabei in der ersten Zeile und in exakt dem folgenden Format notiert werden, da wir sonst nicht automatisiert eine Statistik erheben können:

```
Zeitaufwand: 3:30

<...Andere Erfahrungen...>
```

Die Angabe `3:30` steht hier für 3 Stunden und 30 Minuten.

[1] Die Zeit kann dabei einfach anhand der Schleifendurchläufe und der mit `usleep` erzwungenen Wartezeit approximiert werden. Im ersten Schleifendurchlauf der `plot`-Funktion hat die Zeit dann Wert `0.0`.