**Institut für Wissenschaftliches Rechnen**
**Technische Universität Braunschweig**
Prof. Hermann G. Matthies, Ph. D.
Dr. Elmar Zander

Sommersemester 2010
14. November 2014

# Weiterführendes Programmieren

## *Datentypen*
## Aufgabenblatt 3

Auf diesem Aufgabenblatt geht es in erster Linie um Datenstrukturen. Wir werden zunächst in der ersten Aufgabe eine Datenstruktur zur Repräsentation rationaler Zahlen besprechen. Da jede rationale Zahl die gleiche Form hat, nämlich einen Zähler und einen Nenner besitzt, kann man Sie mit einer einfachen *statischen* Struktur im Speicher repräsentieren. Datenstrukturen werden in C mit `struct`-Deklarationen beschrieben. Dieses Konstrukt wird ihnen in den Aufgaben begegnen.

In der zweiten Aufgabe werden Sie eine dynamische (also nicht-statische) Datenstruktur umsetzen müssen. Dynamische Datenstrukturen werden immer dann eingesetzt, wenn nicht von Vornherein klar ist, wie die Struktur der Daten im Speicher aussieht. Die Länge einer Liste ist z.B. nicht bei allen Listen gleich; eine Liste kann daher nicht als statische, sondern muss als dynamische Datenstruktur umgesetzt werden. Damit Sie auch diese Datenstrukturen kennenlernen, werden Sie einen Zahlenstapel (Stack) implementieren, der die im ersten Teil programmierten rationalen Zahlen abspeichern kann.

Mit Hilfe dieses Stapels werden Sie dann in der dritten Aufgabe einen Stapeltaschenrechner umsetzen, der mit ihrer Implementierung der Arithmetik auf rationalen Zahlen arbeiten soll.

**Übung 1:** *Rationale Zahlen*
Da wir in dieser Aufgabe zum ersten mal mit Strukturen umgehen werden, sollten Sie Kapitel 11.1 (13 Seite) im Buch [Dausmann,Bröckl,Goll:2008] lesen. Außerdem werden wir hier ein Array benutzen, daher sollten Sie auch die Kapitel 6, 10.1 und 10.2 (16+17 Seiten) im Buch [Dausmann,Bröckl,Goll:2008] gelesen haben.
Bitte laden Sie sich zunächst alle Dateien von:
`http://www.wire.tu-bs.de/ADV/files/rational/`
herunter und speichern Sie diese in einem Verzeichnis mit dem Namen `rational`.
In der Headerdatei `rational.h` finden Sie nun alle nötigen Informationen, um die in der Datei `rational.c` unvollständig gegebenen Funktionen zu vervollständigen. Um Ihre Implementierung zu testen, benutzen Sie die Datei `rational_test.c`, welche einen Test für die rationalen Zahlen enthält. Den Test kompilieren Sie am besten einfach mit einem Aufruf von `make`, welcher mit dem mitgelieferten `Makefile` automatisch den Test übersetzt. Starten Sie den Test dann wie gewöhnlich mit `./rational_test`.
**Aufgabe a)** Vervollständigen Sie die Implementierung der rationalen Zahlen in der Datei `rational.c` und benutzen Sie den Test, um zu zeigen, dass Ihre Umsetzung den Vorgaben aus der Datei `rational.h` entspricht. □

**Übung 2:** *Abstrakte Datentypen*

In dieser Aufgabe geht es zunächst darum einen Stapel (Englisch: stack) für `double`-Werte zu implementieren. Dieser Stapel soll dann im zweiten Teil der Aufgabe auf einen Stapel für rationale Zahlen umgestellt werden.

Das klingt komplizierter als es ist, denn Stapel kennen wir aus unserem alltäglichen Leben, wenn wir z.B. Teller stapeln. Eine Stapel-Struktur bietet zwei Funktionen an, mit denen man den Zustand des Stapels verändern kann. Man kann Dinge auf den Stapel legen oder das oberste Element wieder abheben. Bei einem Tellerstapel kann man zwar auch Teller aus der Mitte herausziehen, in der formalen Spezifikation des Stapel ist das aber nicht vorgesehen.

Eine mehr technische Sichtweise auf den Stapel ist, den Stapel als Speicher zu betrachten, der eine bestimmte Zugriffsregel anbietet: Man bekommt das Element, das man als letztes gespeichert hat, als erstes wieder heraus. Aus diesem Grund wird der Stapel technisch auch als *LIFO* [1] (Last in First out) bezeichnet.

Wenn Sie mehr über den Stapel als Datenstruktur wissen wollen, lesen Sie `http://de.wikipedia.org/wiki/Stapelspeicher`.

Nun zu der etwas formaleren Spezifikation eines Stapel. Wir geben Ihnen eine Headerdatei vor:

```
1  #ifndef __STACK_H
2  #define __STACK_H
3  #define true  1 //we can use true instead of 1
4  #define false 0 //we can use false instead of 0
5
6  /*****************************************************************************/
7  /* name       : Stack                                                        */
8  /* Description: Data-structure to represent a Stack for double values    */
9  /*****************************************************************************/
10   typedef struct double_stack_head_struct
11   {
12     struct double_stack_head_struct* tail;
13     double                           value;
14   } double_stack_head;
15
16   // the structure containing the state of a stack
17   typedef struct double_stack_struct
18   {
19     int         size;     // size of stack
20     double_stack_head* head;
21   } double_stack;
22
23 /*****************************************************************************/
24 /* name    : init_stack                                                      */
25 /* args    : pointer to a Stack, which is to be initialised,                 */
26 /* return  : void                                                            */
27 /* function: delete stack if already allocated; initialise structure     */
28 /*****************************************************************************/
29   void init_double_stack(double_stack* stack);
30
```

[1] Sie haben vielleicht auch schon mal das Worte FIFO (First in First out) gehört. Diese Speicher werden häufig als Buffer eingesetzt.

```
31 /*****************************************************************************/
32 /* name    : delete_stack                                                    */
33 /* args    : pointer to a Stack, which will be deleted                       */
34 /* return  : void                                                            */
35 /* function: delete reserved memory if allocated                             */
36 /*           set size to 0, head to NULL                                     */
37 /*****************************************************************************/
38   void delete_double_stack(double_stack* stack);
39
40 /*****************************************************************************/
41 /* name    : empty_stack                                                     */
42 /* args    : pointer to a constant Stack, which will not be changed here     */
43 /* return  : true if stack is empty, false otherwise                         */
44 /* function: if stack is empty return true; else return false                */
45 /*****************************************************************************/
46   int empty_double_stack(const double_stack* stack);
47
48 /*****************************************************************************/
49 /* name    : push_stack                                                      */
50 /* args    : pointer to stack which is to be pushed with "value"             */
51 /* return  : void                                                            */
52 /* function: push value onto stack                                           */
53 /*****************************************************************************/
54   void push_double_stack(double_stack* stack, double value);
55
56 /*****************************************************************************/
57 /* name    : pop_stack                                                       */
58 /* args    : pointer to stack which is to be poped,                          */
59 /*           pointer to variable to which top value is to be written         */
60 /* return  : true if value has been written to "value"                       */
61 /*           false otherwise                                                 */
62 /* function: if stack is not empty                                           */
63 /*               write to "value" the top value;                             */
64 /*               pop head and use tail as new head                           */
65 /*               return true                                                 */
66 /*           else return false                                               */
67 /*****************************************************************************/
68   int pop_double_stack(double_stack* stack, double* value);
69
70 #endif
```

Listing 1: Headerdatei – `double_stack.h`

**Bemerkung:** Wenn Sie Listing 1 genauer betrachten, sehen Sie, dass die Namen der genannten Funktionen sehr lang sind. In C ist die Tendenz zu solch langen Funktionsnamen durch eine Eigenschaft der Sprache gegeben, die wir Ihnen an dieser stelle kurz näherbringen wollen.

```
1 double _abs(double);
2 int    _abs(int);
```

Listing 2: Fehlerhafte Definition

In Listing 2 werden zwei verschiedene Deklarationen mit dem Symbolnamen `_abs` gemacht. Dies ist nicht legal, da C nur diesen Symbolnamen zur Unterscheidung von Funktionen heranzieht und Zeile 2 daher einen Compilierfehler

```
line 2: error: conflicting types for '_abs'
line 1: error: previous declaration of '_abs' was here
```

ergibt. Die Verwendung solche langer Namen, wie in Listing 1 ist daher sinnvoll, wenn Funktionen für mehrere Typen definiert werden sollen. Der Typname wird dann per Konvention Teil des Funktionsnames.

**Bemerkung:** Die ersten beiden Zeilen zusammen mit der letzten Zeile der Header-Datei haben die Funktion dass Sie einen zweifachen Import der Deklarationen vermeiden und damit einen Compiler-Fehler für den folgenden Code verhindern:

```
1 #include "double_stack.h"
2 #include "double_stack.h"
```

Erinnern Sie sich an diesen Trick, wenn Sie selbst Header-Dateien schreiben.

**Bemerkung:** Jede Deklaration in der Header-Datei ist mit einem Block von Kommentaren kombiniert. Das ist guter Programmierstil, da der Kommentar nicht nur dazu dient, dass Sie wissen, was Sie Programmieren müssen, sondern auch Anwendern dazu dient, zu verstehen, wie man den Stack benutzt.

Technisch kann man einen Stapel auf sehr unterschiedliche Weisen umsetzen, wir haben uns in dieser Aufgabe dazu entschlossen eine einfach verkettete Liste (siehe `http://de.wikipedia.org/wiki/Liste_(Datenstruktur)`) zu nutzen, in der immer vorne neue Einträge an die Liste angefügt werden. Die erforderliche Speicherstruktur ist in den Zeilen 10-20 im Listing 1 gegeben. In Abbildung 1 ist die Verwendung der Struktur an einem Beispiel dargestellt.

**Aufgabe a)** Laden Sie von der Webseite die Vorlagen herunter (unter `http://www.wire.tu-bs.de/ADV/files/double_stack/`) und implementieren Sie die fehlenden Funktionen in `double_stack.c`.
□

**Aufgabe b)** Testen Sie Ihre Implementierung mit dem mitgelieferten Test `double_stack_test.c`.
□

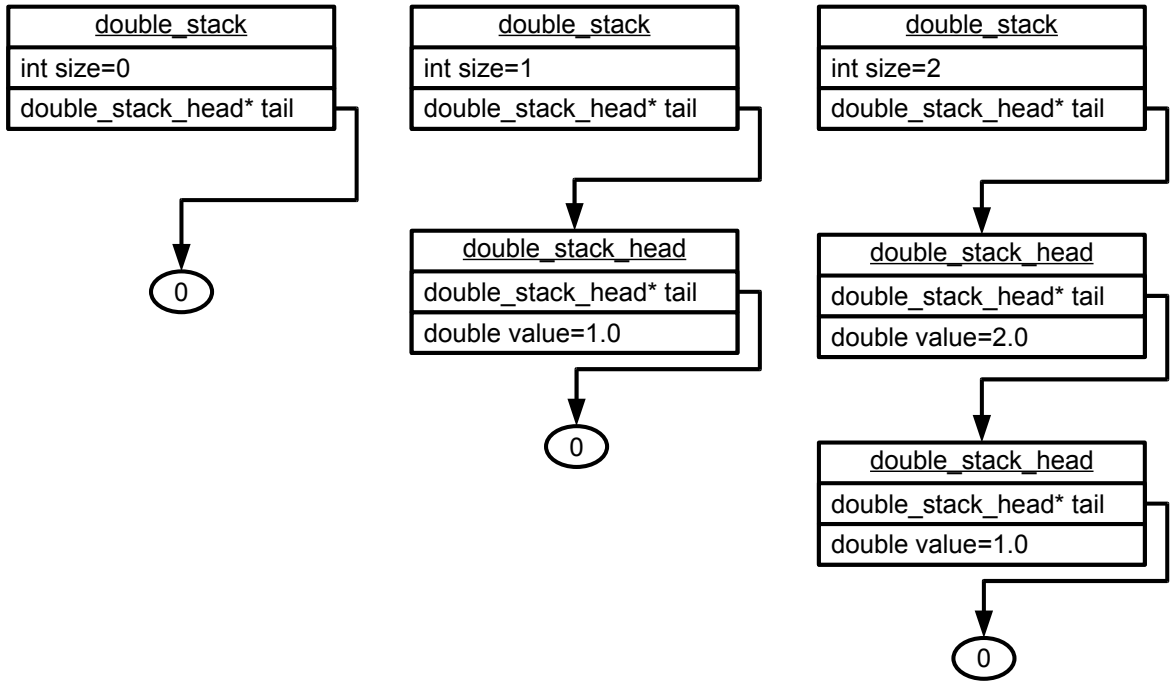


Abbildung 1: Das Bild illustriert anhand von drei Beispielen, wie ein Stack mit Hilfe der in Listing 1 aufgeführten Struktur aufgebaut werden kann. In dem Beispiel wird (von links nach rechts) auf einen leeren Stack zunächst der Wert 1.0, dann der Wert 2.0 “gepushed”.

**Aufgabe c)** Implementieren Sie einen Stack für die in der ersten Aufgabe auf diesem Blatt implementierten rationalen Zahlen. Behalten Sie dabei die Namenskonvention für Dateien und Funktionen bei, nennen Sie z.B. die Headerdatei also `rational_stack.h`. □

**Übung 3:** *Rechner mit **U**mgekehrt **P**olnischer **N**otation (UPN)*

Auf `http://de.wikipedia.org/wiki/Umgekehrte_Polnische_Notation` können Sie nachlesen, was man unter umgekehrt polnischer Notation (UPN) versteht. Hier soll ein *Taschenrechner* entwickelt werden, der diese Notation mit rationalen Zahlen umsetzt.

Für diese Aufgabe liefern wir Ihnen unter `http://www.wire.tu-bs.de/ADV/files/stack_calc` eine Vorlage, welche einen UPN-Taschenrechner mit Fließkommazahlen implementiert. Um das gegebene Programm zum Laufen zu bringen, müssen Sie noch ihre getestete Implementierung des `double_stack` in das Verzeichnis kopieren (ohne das Makefile!!!).

**Aufgabe a)** Verändern Sie den gegebenen UPN-Rechner so, dass er mit den rationalen Zahlen aus der ersten Aufgabe funktioniert. Dazu schauen Sie sich die gegebene, gut kommentierte Vorlage in Ruhe an. Eine rationale Zahl soll dabei mit dem Operator ’|’, also dem *senkrechten Strich* definiert werden. In UPN also: 1 2 | □