Programmentwicklung II

Bachelor of Science

Prof. Dr. Rethmann / Prof. Dr. Brandt

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

Sommersemester 2021

Wiederholung C

- Strukturen
- Arrays
- Pointer
- Zeichenketten
- Aufteilen des Codes auf verschiedene Dateien
- dynamische Speicheranforderung
- Rekursion

Programm 1

Addieren zweier Brüche

main.c

```c
#include <stdio.h>
#include "bruch.h"

int main(void) {
    bruch_t a = {1, 2};
    bruch_t b = {3, 4};
    bruch_t c = add(a, b);

    output(c);
    return 0;
}
```

$$\frac{1}{2} + \frac{3}{4} = \frac{1 \cdot 4 + 3 \cdot 2}{2 \cdot 4} = \frac{10}{8} = \frac{5}{4}$$

bruch.h

```
#ifndef _BRUCH_H
#define _BRUCH_H

typedef struct {
    int z, n;
} bruch_t;

bruch_t add(bruch_t a, bruch_t b);
void output(bruch_t q);

#endif
```

bruch.c

```c
#include "bruch.h"

static   // von außen nicht sichtbar !!!
int gcd(int p, int q) {
    int r;
    do {
        r = p % q;
        p = q;
        q = r;
    } while (r != 0);
    return p;
}

static   // von außen nicht sichtbar !!!
void reduce(bruch_t *q) {  // warum ein Zeiger als Parameter ??
    int t = gcd(q->z, q->n);
    q->z /= t;
    q->n /= t;
}
```

Addieren zweier Brüche

```
bruch_t add(bruch_t a, bruch_t b) {
    int z = a.z * b.n + b.z * a.n;
    int n = a.n * b.n;

    bruch_t r = {z, n};
    reduce(&r);  // call by reference !!!

    return r;
}
```

Übersetzen:

```
gcc -Wall -Wextra -O3 -c main.c
gcc -Wall -Wextra -O3 -c bruch.c
gcc main.o bruch.o -o test
```

Programm 2

main.c

```c
#include <stdio.h>
#include "bruch.h"

int main(void) {
    // ein Array von fünf Brüchen:
    bruch_t werte[5] = {{1,2}, {1,3}, {1,4}, {1,5}, {1,6}};
    bruch_t result = summe(werte, 5);  // Array als Parameter

    char s[32], str[5][32];  // Zeichenketten
    for (int i = 0; i < 5; i++)
        btos(werte[i], str[i], 32);  // bruch-to-string
    btos(result, s, 32);

    printf("%s + %s + %s + %s + %s = %s\n", str[0], str[1],
           str[2], str[3], str[4], s);
    return 0;
}
```

bruch.h

```
#ifndef _BRUCH_H
#define _BRUCH_H

typedef struct {
    int z, n;
} bruch_t;

bruch_t add(bruch_t a, bruch_t b);
void output(bruch_t q);
bruch_t summe(bruch_t *values, int n);  // neu
void btos(bruch_t q, char *s, int n);   // neu

#endif
```

```
....  // wie oben

bruch_t summe(bruch_t *values, int n) {
    bruch_t r = values[0];
    for (int i = 1; i < n; i++)
        r = add(r, values[i]);
    return r;
}

void btos(bruch_t q, char *s, int n) {
    snprintf(s, n, "(%d/%d)", q.z, q.n);
}
```

bruch.c

Programm 3

aus der Mathematik[1]:

$$\sqrt{2} = 1 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \dots}}}} \qquad \pi = \cfrac{4}{1 + \cfrac{1^2}{3 + \cfrac{2^2}{5 + \cfrac{3^2}{7 + \dots}}}}$$

als Struktur in C:

```
typedef struct kettenbruch_s {
    int z, n;
    struct kettenbruch_s *c;
                    // ^^^ warum ein Zeiger ???
} kettenbruch_t;
```

[1] https://de.wikipedia.org/wiki/Kettenbruch

main.c

```c
#include <stdio.h>
#include "kette.h"

int main(void) {
    kettenbruch_t k5 = {1, 2, NULL};
    kettenbruch_t k4 = {1, 2, &k5};
    kettenbruch_t k3 = {1, 2, &k4};
    kettenbruch_t k2 = {1, 2, &k3};
    kettenbruch_t k1 = {1, 2, &k2};
    printf("1 + eval(k1) = %.9lf\n", 1 + eval(&k1));
    return 0;
}
```

kette.h

```
#ifndef _KETTE_H
#define _KETTE_H

typedef struct kettenbruch_s {
    int z, n;
    struct kettenbruch_s *c;
} kettenbruch_t;

double eval(kettenbruch_t *c);

#endif
```

kette.c

```c
#include "kette.h"

double eval(kettenbruch_t *b) {
    if (b->c == NULL)
        return (double) b->z / b->n;    // warum type-cast?
    return b->z / (b->n + eval(b->c)); // warum kein type-cast?
}
```

Programm 4

main.c

```c
#include <stdio.h>
#include "bruch.h"

int main(void) {
    bruch_t *a = getRational(1, 2);
    bruch_t *b = getRational(3, 4);
    bruch_t *c = add(a, b);

    output(c);
    cleanRational(a);
    cleanRational(b);
    // cleanRational(c); !!!!!
    return 0;
}
```

bruch.h

```
#ifndef _BRUCH_H
#define _BRUCH_H

// incomplete data type or forward declaration
typedef struct bruch_s bruch_t;

// public interface
bruch_t *getRational(int z, int n);
void cleanRational(bruch_t *q);

bruch_t *add(bruch_t *a, bruch_t *b);
void output(bruch_t *q);
void btos(bruch_t *q, char *s, int n);
bruch_t *summe(bruch_t **values, int n);
                    // ^^^ warum 2 Zeiger ?????

#endif
```

bruch.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "bruch.h"

struct bruch_s {
    int z, n;
};

bruch_t *getRational(int z, int n) {
    bruch_t *r = (bruch_t *) malloc(sizeof(bruch_t));
    r->z = z;
    r->n = n;
    return r;
}

void cleanRational(bruch_t *q) {
    free(q);
}
```

dynamische Speicheranforderung

```
bruch_t *add(bruch_t *a, bruch_t *b) {
    static bruch_t r;  // warum static ?????

    r.z = a->z * b->n + b->z * a->n;
    r.n = a->n * b->n;
    reduce(&r);
    return &r;
}

bruch_t *summe(bruch_t **values, int n) {
    static bruch_t r;  // warum static ?????

    r = *values[0];
    for (int i = 1; i < n; i++)
        r = *add(&r, values[i]);  // Syntax ?????
    return &r;
}

...  // wie oben
```

Programm 5

main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "kette.h"

int main(void) {
    kettenbruch_t *ke = getChainFrac(1, 2, NULL);
    kettenbruch_t *k = getChainFrac(1, 2, ke);
    for (int i = 0; i < 10; i++) {
        k = getChainFrac(1, 2, k);
        printf("1 + eval(k) = %.9lf\n", 1 + eval(k));
    }
    printf("    sqrt(2) = %.9lf\n", sqrt(2));
    return 0;
}
```

kette.h

```
#ifndef _KETTE_H
#define _KETTE_H

// incomplete data type or forward declaration
typedef struct kettenbruch_s kettenbruch_t;

kettenbruch_t * getChainFrac(int z, int n, kettenbruch_t *c);
double eval(kettenbruch_t *c);

#endif
```

kette.c

```c
#include "kette.h"

struct kettenbruch_s {
    int z, n;
    struct kettenbruch_s *c;
};

kettenbruch_t * getChainFrac(int z, int n, kettenbruch_t *c) {
    kettenbruch_t *r = (kettenbruch_t *)
                       malloc(sizeof(kettenbruch_t));
    r->z = z;
    r->n = n;
    r->c = c;
    return r;
}

double eval(kettenbruch_t *c) {
    // wie oben
}
```

Übung

- Schreiben Sie eine Funktion `cleanChainFrac`, die den durch einen Kettenbruch belegten Speicher wieder frei gibt.
- Schreiben Sie ein Programm, dass den Wert von π mittels eines Kettenbruchs approximiert.