1 Allgemein

Folgend ist der Teil der Dokumentation angeführt, welche die Programmstruktur sowie das Ausführen der Tests dokumentiert.

1.1 Lösungsidee

Folgend ist dokumentiert, wie die Programmstruktur aufgebaut ist.

Programmstruktur:

1. weight/
   Das Verzeichnis, welches den Source, Header und makefile für das Programm weight enthält
2. weight/test/
   Das Verzeichnis, welches das Testskript für das Programm weight enthält.
3. polynom/
   Das Verzeichnis, welches den Source, Header und makefile für das Programm polynom enthält
4. polynom/test/
   Das Verzeichnis, welches das Testskript für das Programm polynom enthält.

Ausführen der Testskripten:

1. Entpacken Sie das Archiv auf einen Linux System
2. Navigieren Sie in das Verzeichnis, wo das Shell Script enthalten ist.
   (<archive_root>/<program_name>/test)
3. Stellen Sie sicher das der Benutzer ausreichende Rechte für das entpackte Verzeichnis und seinen Inhalt hat
4. Machen Sie die Skripten ausführbar. `chmod +x *_tests.sh`
5. Führen Sie die Testskripten aus. Das aktuelle Verzeichnis muss *<archive_root>/<program_name>/test* sein, da die Tests sonst nicht funktionieren, da relative Pfade enthalten sind.

2 Krafttraining

Folgend ist die Dokumentation für die Aufgabe Krafttraining angeführt, welche eine Ermittlung von der Anzahl von bestimmten Hantelscheiben (unterschiedliches Gewicht) in Abhängigkeit von dem verlangten Gewicht zu ermitteln.

2.1 Lösungsidee

Folgend ist die Lösungsidee für das Programm weight angeführt, welches die Ermittlung von der Anzahl von bestimmten Hantelscheiben (verschiedenes Gewicht) verlangt, welche verwendet werden können um auf eine Hantelstange ein bestimmtes Gewicht erreichen.

Bei der Initialisierung des Programm soll geprüft werden ob die übergebenen Argumente folgende Eigenschaften erfüllen:

1. Exakt ein Argument
2. Argument kann in Datentyp Double geparst werden (strtod verwenden und nicht atof)
3. Argument ist in folgenden Grenzen
   (BAR_WEIGHT <= argument <= MAX_UNIT_WEIGHT)

Sollte eine dieser Eigenschaften nicht erfüllt sein, so soll eine entsprechende Meldung ausgegeben werden.

Globale Variablen:

1. UNITS
   Statisch konstantes Array mit den verfügbaren Gewichten
2. UNIT_COUNTS
   Statisch konstantes Array mit der Anzahl der verfügbaren Gewichten
3. MAX_UNITS
   Die Anzahl der maximal verfügbaren Gewichten
4. BAR
   Statische Konstante, die das Gewicht der Stange definiert
5. solutions
   Statisches Array der gefunden Lösungen
6. solutionCount
   Der Zähler für die gefundenen Lösungen

Das Programm soll so weit wie möglich unterteilt werden und nicht der gesamte Code in einer Funktion enthalten sein. Alle Funktionsdeklarationen sollen in einer eigenen Header Datei ausgelagert werden, sowie auch alle verwendeten Konstanten und globalen Variablen. Der Hauptalgorithmus soll über Backtracking die Lösungen ermitteln, wobei hierbei alle Lösungen durchprobiert werden sollen.

Hierbei wird über alle Units iteriert und diese einem Array hinzugefügt, sowie eine Variable, die das Gesamtgewicht hält durch die rekursiv aufgerufene Funktion geschleift werden. Hierbei soll mit Pointern gearbeitet werden.

2.2 Implementierung

Folgend ist der Source die Header Datei und die Makefile des Programms weight angeführt.

2.2.1 weight.h

```
/*
 * This header file contains all constants, makros and function declaration
 * the application 'weights' needs.
 *
 * weight.h
 *
 *  Created on: Sep 28, 2014
 *      Author: Thomas Herzog
 */
#include "common.h"
#include <limits.h>
#ifndef WEIGHT_H_
#define WEIGHT_H_

/* the maximum available units */
#define MAX_UNITS 8

/* The available weights of the units */
static const double UNITS[MAX_UNITS] = { 0.5, 1.25, 2.5, 5, 10, 15, 20, 25 };
/* The available counts of the units */
static const int UNIT_COUNTS[MAX_UNITS] = { 1, 1, 1, 3, 1, 1, 3, 1 };
/* The weight of the bar */
static const double BAR = 20.0;
/* Holds the solutions */
static int solutions[SHRT_MAX][MAX_UNITS];
/* The solution counter */
static int solutionCount = 0;

/**
 * Validates the given program arguments if they are valid arguments for this application.
 *
 * char** arguments: the two dimensional array of arguments
 * double* value: the parsed argument value
 * return bool: true if the arguments are valid, false otherwise
 */
bool ValidateArguments(int argc, char** arguments, double* value);

/**
 * Answers the question if the unit at the index is available to be used or not.
 *
 * usedUnitCount[MAX_UNITS]: the array holding the already used units counts on the index of
the unit
 * int intendedUnitIdx: the index of the to add unit
 * return bool: true if useable, false if all counts are already used
 */
bool IsUnitAvailable(int usedUnitCount[MAX_UNITS], int intendedUnitIdx);

/**
 * Adds the unit at the index to the array and sums the unit weight to the sumWeight.
 *
 * int (*usedUnitCount)[MAX_UNITS]: the array holding the used unit count at the index of the
unit
 * double* sumWeight: the summary of the weights already used
 * int idx: the index of the unit to be added
 * return void
 */
void AddUnit(int (*usedUnitCount)[MAX_UNITS], double* sumWeight, int idx);

/**
```

```
 * Removes the unit at the index to the array and subtracts the unit weight to the sumWeight.
 *
 * int (*usedUnitCount)[MAX_UNITS]: the array holding the used unit count at the index of the
unit
 * double* sumWeight: the summary of the weights already used
 * int idx: the index of the unit to be added
 * return void
 */
void RemoveUnit(int (*usedUnitCount)[MAX_UNITS], double* sumWeight, int idx);

/**
 * Inits the used array by setting all counts to '0'.
 *
 * int (*usedUnitCount)[MAX_UNITS]: the array holding the used unit count at the index of the
unit
 */
void Init(int (*usedUnitCount)[MAX_UNITS]);

/**
 * Prints the solutions only once. it keeps the already found solutions in a static function
visible variable.
 *
 * int usedUnitCount[MAX_UNITS]: the array holding the used unit count at the index of the
unit
 */
void PrintSolution(int usedUnitCount[MAX_UNITS]);

/**
 * Tries all possibilities to find combination of the units which represent the intended
weight.
 *
 * int i: the current index in the array which holds the unit counts
 * int (*usedUnitCount)[MAX_UNITS]: the array holding the used unit count at the index of the
unit
 * double* sumWeight: the summary of the weights
 * double* intendedWeight: the intended weight
 */
void GetUnitCombination(int i, int (*usedUnitCount)[MAX_UNITS],
        double* sumWeight, double* intendedWeight);

#endif /* WEIGHT_H_ */
```

2.2.2 weight.c

```
/*
 * This program detects the unit combination for a certain weight.
 * All of the solutions will be detected and print to the console.
 * weight.c
 *
 *  Created on: Sep 28, 2014
 *      Author: Thomas Herzog
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "weight.h"

int main(int argc, char** argv) {
    int usedUnitCount[MAX_UNITS];
    int i;
    double sumWeight;
    double value;

    /* Exit if arguments are invalid */
    if (!ValidateArguments(argc, argv, &value)) {
        return EXIT_FAILURE;
    } /* if */

    /* calculate unit combination */
    i = 0;
    sumWeight = 0;
    value -= BAR;
    Init(&usedUnitCount);
```

```
    GetUnitCombination(i, &usedUnitCount, &sumWeight, &value);

    if (solutionCount == 0) {
        printf(
                "Sorry no unit combination found for your intended weight found !!!");
    } /* if */
    return EXIT_SUCCESS;
}

void Init(int (*usedUnitCount)[MAX_UNITS]) {
    int i;
    for (i = 0; i < MAX_UNITS; ++i) {
        (*usedUnitCount)[i] = 0;
    } /* for */
}

void PrintSolution(int usedUnitCount[MAX_UNITS]) {
    int i, j;
    bool alreadyFound = false;

    /* Find out if solution already found over all found solutions */
    i = 0;
    while ((i < solutionCount) && (!alreadyFound)) {
        j = 0;
        alreadyFound = true;
        /* If solution not at index i then break search */
        while ((j < MAX_UNITS) && (alreadyFound)) {
            if (solutions[i][j] != usedUnitCount[j]) {
                alreadyFound = false;
            } /* if */
            j++;
        } /* while */
        i++;
    } /* while */

    /* If not already found add the newly found solution */
    if (!alreadyFound) {
        solutionCount += 1;
        printf("--------------\n");
        printf("Solution %d:\n", solutionCount);
        printf("--------------\n");
        for (i = 0; i < MAX_UNITS; ++i) {
            solutions[(solutionCount - 1)][i] = usedUnitCount[i];
            if (usedUnitCount[i] != 0) {
                printf("unit %d | weight: %6.2f | count: %d pair\n", i,
                        UNITS[i], usedUnitCount[i]);
            } /* if */
        } /* for */
        printf("\n");
    } /* if */
}

void GetUnitCombination(int i, int (*usedUnitCount)[MAX_UNITS],
        double* sumWeight, double* intendedWeight) {
    int j, k, m;
    /* Iterate over all remaining units */
    for (j = i; j < MAX_UNITS; ++j) {
        /* Check if at least one unit count is left */
        if (IsUnitAvailable(*usedUnitCount, j)) {
            /* Iterate over the remaining unit UNIT COUNTS */
            for (k = (*usedUnitCount)[j]; k < UNIT_COUNTS[j]; k++) {
                /* Add the unit count */
                AddUnit(usedUnitCount, sumWeight, j);
                /* If summary fits intended solution found */
                if (*sumWeight == *intendedWeight) {
                    PrintSolution(*usedUnitCount);
                }
                /* To large need to go back on unit count */
                else if (*sumWeight > *intendedWeight) {
                    RemoveUnit(usedUnitCount, sumWeight, j);
                }
                /* Otherwise keep trying */
                else {
                    GetUnitCombination((i + 1), usedUnitCount, sumWeight,
                            intendedWeight);
                } /* if */
```

```
            } /* for */
            /* Remove added units */
            for (m = 0; m < (*usedUnitCount)[j]; ++m) {
                RemoveUnit(usedUnitCount, sumWeight, j);
            } /* for */
        } /* if */
    } /* for */
}

void AddUnit(int (*usedUnitCount)[MAX_UNITS], double* sumWeight, int idx) {
    /* Increase count and weight of used unit */
    if ((*usedUnitCount)[idx] < UNIT_COUNTS[idx]) {
        *sumWeight += (UNITS[idx] * 2);
        (*usedUnitCount)[idx] += 1;
    }
}

void RemoveUnit(int (*usedUnitCount)[MAX_UNITS], double* sumWeight, int idx) {
    /* Decrease unit count and weight of removed unit */
    if ((*usedUnitCount)[idx] > 0) {
        *sumWeight -= (UNITS[idx] * 2);
        (*usedUnitCount)[idx] -= 1;
    } /* if */
}

bool IsUnitAvailable(int usedUnitCount[MAX_UNITS], int idx) {
    return (UNIT_COUNTS[idx] > usedUnitCount[idx]) ? true : false;
}

bool ValidateArguments(int argc, char** argv, double* value) {
    char* tail = "";
    int i = 0;
    double maxValue = 0;

    /* Handle invalid argument count */
    if (argc != 2) {
        printf(
                "Program accepts exactly one argument of type double !!! argc: %d",
                (argc - 1));
        return false;
    } /* if */

    /* Check if argument is valid double argument */
    *value = strtod(argv[1], &tail);
    if (strlen(tail) != 0) {
        printf("Argument is no valid double !!! argument: %s", argv[1]);
        return false;
    } /* if */

    /* Check if argument is in valid range */
    for (i = 0; i < MAX_UNITS; ++i) {
        maxValue += (UNITS[i] * (UNIT_COUNTS[i] * 2));
    } /* for */

    /* Value violates range */
    if ((*value < BAR) || (*value > maxValue)) {
        printf(
                "Requested weight is not available !!! BAR: %3.2f | max-unit-weight: %3.2f |
requested: %3.2f",
                BAR, maxValue, *value);
        return false;
    }
    /* weight equals used BAR */
    else if (*value == BAR) {
        printf(
                "Requested weight equals BAR weight, so need to add any units\n");
        return false;
    } /* if */

    /* everything is ok */
    return true;
}
```

2.2.3 weight_tests.sh

```
#!/bin/sh

## Varaibles used during build and tests ##
BINARY=weight

## Copy all stuff together into test folder ##
echo "############################################"
echo "This script tests the program weight"
echo "############################################"
echo ""
echo "creating test resource structure ..."
cd ..
if [ ! -d "weight_tests" ];
then
    mkdir weight_tests
    echo "creating directory 'weight_tests'"
else
    rm -f weight_tests/*
    echo "cleaning directory 'weight tests'"
fi
cd weight_tests
echo "in directory $(pwd)"
echo "copying header, source and make files"
cp ../source/weight.c weight.c
cp ../header/weight.h weight.h
cp ../header/common.h common.h
cp ../make/makefile makefile

## Compile source via make ##
echo "invoking make targets 'clean' 'depend' '$BINARY'\n\n"
make clean depend $BINARY

## Test Invalid arguments ##
echo "-----------------------------------------"
echo "Test with invalid arguments"
echo "-----------------------------------------"
echo "Case 1: ./$BINARY [no argument]"
./$BINARY
echo "\n"

echo "Case 2: ./$BINARY 1 2.0 [too many argument]"
./$BINARY 1 2.0
echo "\n"

echo "Case 3: ./$BINARY a [argumetn not valid double]"
./$BINARY a
echo "\n"

echo "Case 4: ./$BINARY 1.a [argumetn not valid double]"
./$BINARY 1.a
echo "\n"

echo "Case 5:  ./$BINARY 1,2 [argumetn not valid double]"
./$BINARY 1,2
echo "\n"

echo "Case 7: ./$BINARY -1 [out of range]"
./$BINARY -1
echo "\n"

echo "Case 6: ./$BINARY 9 [out of range]"
./$BINARY 9
echo "\n"

echo "Case 7: ./$BINARY 500 [out of range]"
./$BINARY 500
echo "\n"

## Test valid arguments ##
echo "-----------------------------------------"
echo "Test with valid arguments"
echo "-----------------------------------------"
echo "Case 1: ./$BINARY 20 [Exact as bar weight]"
./$BINARY 20
```

```
echo "\n"

echo "Case 2: ./$BINARY 21 [found solution]"
./$BINARY 21
echo "\n"

echo "Case 3: ./$BINARY 251 [found solution]"
./$BINARY 251
echo "\n"

echo "Case 4: ./$BINARY 61 [found solution]"
./$BINARY 61
echo "\n"

echo "Case 5: ./$BINARY 158.5 [found solution]"
./$BINARY 158.5
echo "\n"

echo "Case 6: ./$BINARY 36 [found solution]"
./$BINARY 36
echo "\n"

echo "Case 7: ./$BINARY 72.5 [found solution]"
./$BINARY 72.5
echo "\n"

echo "Case 8: ./$BINARY 33.3 [no solution]"
./$BINARY 33.3
echo "\n"

echo "Case 9: ./$BINARY 30.75 [no solution]"
./$BINARY 30.75
echo "\n"

echo "Case 10: ./$BINARY 120.80 [no solution]"
./$BINARY 120.80
echo "\n"

## Cleanup created test resources ##
echo "cleanup test"
cd ..
rm -rf weight_tests
echo "cleanup complete"
```

2.3 Tests

Folgend sind die Test des Programms weight angeführt.

2.3.1 Ungültige Argumente

Folgend sind die Test für ungültige Argumente angeführt.

```
-----------------------------------------
Test with invalid arguments
-----------------------------------------
Case 1: ./weight [no argument]
Program accepts exactly one argument of type double !!! argc: 0

Case 2: ./weight 1 2.0 [too many argument]
Program accepts exactly one argument of type double !!! argc: 2

Case 3: ./weight a [argumetn not valid double]
Argument is no valid double !!! argument: a

Case 4: ./weight 1.a [argumetn not valid double]
Argument is no valid double !!! argument: 1.a

Case 5:  ./weight 1,2 [argumetn not valid double]
Argument is no valid double !!! argument: 1,2

Case 7: ./weight -1 [out of range]
Requested weight is not available !!! BAR: 20.00 | max-unit-weight: 258.50 | requested: -1.00

Case 6: ./weight 9 [out of range]
Requested weight is not available !!! BAR: 20.00 | max-unit-weight: 258.50 | requested: 9.00

Case 7: ./weight 500 [out of range]
Requested weight is not available !!! BAR: 20.00 | max-unit-weight: 258.50 | requested: 500.00
```

Sollte die Argumentanzahl, Argumenttyp nicht mit der Spezifikation übereinstimmen oder der vorgeschriebener Wertebereich über- oder unterschritten werden, so wird dies dem Aufrufer mitgeteilt.

2.3.2 Gültige Argumente

Folgend sind die Tests angeführt, die gültige Argumente testen.

```
-----------------------------------------
Test with valid arguments
-----------------------------------------
Case 1: ./weight 20 [Exact as bar weight]
Requested weight equals BAR weight, so need to add any units


Case 2: ./weight 21 [found solution]
--------------
Solution 1:
--------------
unit 0 | weight:   0.50 | count: 1 pair



Case 3: ./weight 251 [found solution]
--------------
Solution 1:
--------------
unit 0 | weight:   0.50 | count: 1 pair
unit 3 | weight:   5.00 | count: 3 pair
unit 5 | weight:  15.00 | count: 1 pair
unit 6 | weight:  20.00 | count: 3 pair
unit 7 | weight:  25.00 | count: 1 pair

--------------
Solution 2:
--------------
unit 0 | weight:   0.50 | count: 1 pair
unit 3 | weight:   5.00 | count: 1 pair
unit 4 | weight:  10.00 | count: 1 pair
unit 5 | weight:  15.00 | count: 1 pair
unit 6 | weight:  20.00 | count: 3 pair
unit 7 | weight:  25.00 | count: 1 pair


Case 4: ./weight 61 [found solution]
--------------
Solution 1:
--------------
unit 0 | weight:   0.50 | count: 1 pair
unit 3 | weight:   5.00 | count: 1 pair
unit 5 | weight:  15.00 | count: 1 pair

--------------
Solution 2:
--------------
unit 0 | weight:   0.50 | count: 1 pair
unit 3 | weight:   5.00 | count: 2 pair
unit 4 | weight:  10.00 | count: 1 pair
```

```
Case 5: ./weight 158.5 [found solution]
--------------
Solution 1:
--------------
unit 0 | weight:   0.50 | count: 1 pair
unit 1 | weight:   1.25 | count: 1 pair
unit 2 | weight:   2.50 | count: 1 pair
unit 3 | weight:   5.00 | count: 1 pair
unit 5 | weight:  15.00 | count: 1 pair
unit 6 | weight:  20.00 | count: 1 pair
unit 7 | weight:  25.00 | count: 1 pair

--------------
Solution 2:
--------------
unit 0 | weight:   0.50 | count: 1 pair
unit 1 | weight:   1.25 | count: 1 pair
unit 2 | weight:   2.50 | count: 1 pair
unit 3 | weight:   5.00 | count: 1 pair
unit 6 | weight:  20.00 | count: 3 pair

--------------
Solution 3:
--------------
unit 0 | weight:   0.50 | count: 1 pair
unit 1 | weight:   1.25 | count: 1 pair
unit 2 | weight:   2.50 | count: 1 pair
unit 3 | weight:   5.00 | count: 2 pair
unit 4 | weight:  10.00 | count: 1 pair
unit 6 | weight:  20.00 | count: 1 pair
unit 7 | weight:  25.00 | count: 1 pair

--------------
Solution 4:
--------------
unit 0 | weight:   0.50 | count: 1 pair
unit 1 | weight:   1.25 | count: 1 pair
unit 2 | weight:   2.50 | count: 1 pair
unit 3 | weight:   5.00 | count: 2 pair
unit 5 | weight:  15.00 | count: 1 pair
unit 6 | weight:  20.00 | count: 2 pair

--------------
Solution 5:
--------------
unit 0 | weight:   0.50 | count: 1 pair
unit 1 | weight:   1.25 | count: 1 pair
unit 2 | weight:   2.50 | count: 1 pair
unit 3 | weight:   5.00 | count: 3 pair
unit 4 | weight:  10.00 | count: 1 pair
unit 5 | weight:  15.00 | count: 1 pair
unit 7 | weight:  25.00 | count: 1 pair

--------------
Solution 6:
--------------
unit 0 | weight:   0.50 | count: 1 pair
unit 1 | weight:   1.25 | count: 1 pair
unit 2 | weight:   2.50 | count: 1 pair
unit 3 | weight:   5.00 | count: 3 pair
unit 4 | weight:  10.00 | count: 1 pair
unit 6 | weight:  20.00 | count: 2 pair
```

```
Case 6: ./weight 36 [found solution]
--------------
Solution 1:
--------------
unit 0 | weight:   0.50 | count: 1 pair
unit 2 | weight:   2.50 | count: 1 pair
unit 3 | weight:   5.00 | count: 1 pair


Case 7: ./weight 72.5 [found solution]
--------------
Solution 1:
--------------
unit 1 | weight:   1.25 | count: 1 pair
unit 3 | weight:   5.00 | count: 3 pair
unit 4 | weight:  10.00 | count: 1 pair

--------------
Solution 2:
--------------
unit 1 | weight:   1.25 | count: 1 pair
unit 3 | weight:   5.00 | count: 1 pair
unit 6 | weight:  20.00 | count: 1 pair

--------------
Solution 3:
--------------
unit 1 | weight:   1.25 | count: 1 pair
unit 3 | weight:   5.00 | count: 2 pair
unit 5 | weight:  15.00 | count: 1 pair

--------------
Solution 4:
--------------
unit 1 | weight:   1.25 | count: 1 pair
unit 4 | weight:  10.00 | count: 1 pair
unit 5 | weight:  15.00 | count: 1 pair

--------------
Solution 5:
--------------
unit 1 | weight:   1.25 | count: 1 pair
unit 7 | weight:  25.00 | count: 1 pair


Case 8: ./weight 33.3 [no solution]
Sorry no unit combination found for your intended weight found !!!

Case 9: ./weight 30.75 [no solution]
Sorry no unit combination found for your intended weight found !!!

Case 10: ./weight 120.80 [no solution]
Sorry no unit combination found for your intended weight found !!!
```

Diese Test zeigen, dass wenn verlangte Gewichte mit den zur Verfügung stehenden Gewichten gebildet werden können, dass diese auch korrekt ermittelt werden. Hierbei wird nicht nur eine Lösung sondern alle Lösungen ermittelt und ausgegeben.
Wenn ein Gewicht nicht zur Verfügung steht wird dies dem Aufrufer mitgeteilt.

3 Polynom

Folgend ist die Dokumentation der Aufgabe, welche sich mit Polynomen beschäftigt angeführt.

3.1 Lösungsidee

Folgend ist die Lösungsidee der Aufgabe Polynom angeführt.

3.1.1 Polynomausgabe

Ein Array welches das Polynom darstellt soll auf der Konsole ausgegeben werden wobei besonders darauf zu achten ist, dass die folgende Sonderfälle korrekt behandelt werden.

1. Erstes Element negativ oder 0
2. Nur ein Element an irgendeinem Index vorhanden und negativ
3. Polynomteile im Array sind 0
4. Alle Polynomteile sind 0

Es soll gewährleistet werden, dass diese Ausgaben mit Sonderfällen von dargestellten Polynomen korrekt funktioniert.

3.1.2 Polynomsummation

Hierbei sollen über das Polynom mit dem höchsten Grad iteriert werden, da das Ergebnis maximal n + m an Grad haben kann. Der zuletzt summierte Wert der != 0 ist gibt den Grad des berechneten Polynoms an.

Vorgehensweise:

1. Iteriere über das Polynom mit dem höchsten Grad
2. Summiere die Koeffizienten am Index
   (Achtung dass auf kein Element zugegriffen wird, welches an einem ungültigen Index liegt)
3. Speichere den Grad wenn das Teilergebnis != 0 ist.
4. Führe dies aus bis i == n (höchster Grad) ist

So erhält man das Ergebnis der Summation mit dem Grad des Polynoms, welches die Summe darstellt.

3.1.3 Polynomevaluierung nach x mit Horner Schema

Das Horner Schema für diese Aufgabenstellung soll über eine Schleife, iterativ von hinten nach vorne implementiert werden.
Hierbei wird die Darstellung des Polynoms von hinten weg nach vorne aufgebaut bzw. in dieser Art und Weise berechnet.

```
11 + 7x - 5x^2 -4x^3 + 2x^4 = 11 + x *(7 + x *(-5 + x *(-4 + x * 2)))
                                  ←       ←       ←       ←
```

Vorgehensweise:

1. Initialisiere Variable **result** beim ersten Schleifendurchlauf mit p[n]
2. Multipliziere **result** mit **x**
3. Summiere **result** mit dem nächst niedrigerem Wert **p[i – 1]**
4. Führe dies durch solange **i > 0**

Mit diesem Verfahren sind im oben angeführten Beispiel nur mehr n (= Grad des Polynoms) Multiplikationen von Nöten und keine einzige Potenzrechnung. Mit n – 1 Schleifendurchläufen kann das Polynom nach x berechnet werden.

3.1.4 Polynommultiplikation

Hierbei soll die Multiplikation in 2 Funktionen erfolgen wobei in der ersten Funktion über ein Polynom iteriert wird und das Polynom am Index i mit Grad i in dem zweiten Polynom multipliziert wird, wobei der berechnete Wert am Index j + i (j = Grad des Elements des zweiten Polynoms) im Array abgelegt wird. Es muss darauf geachtet werden dass alle Elemente im Array die zwischen den berechneten Werten mit einem Grad n liegen mit 0 gesetzt werden, damit diese auch initialisiert sind.
Bsp.:
p = 1 + 2x + 3x^2
q = 2 + 3x^2
p * q =

1. 2 * p = 4x + 3x^2 (Hier besteht kein Problem)
2. 3x^2 * p = 3x^2 + 6x^3 + 9x^4
   (Hier müssen i=0, i=1 mit 0 gesetzt werden, da nicht vorhanden im Polynom)

Die berechneten Teile sollen mit der bereits implementierten PolySum Funktion summiert werden. Am Ende erhält man dann das berechnete Produkt.

Vorgehensweise:

1. Iteriere über eines der beiden Polynome
2. Multipliziere den Part am Index i mit dem zweiten Polynom
3. Summiere das Ergebnis mit dem bereits vorherig berechneten Polynomteil oder mit einem leeren Polynom.
   (Achtung beim ersten Mal ist das Array, welches das Ergebnis hält ungültig)
4. Führe dies führ alle Teile des ersten Polynoms durch
5.

Danach erhält man das Produkt der beiden Polynome.

3.1.5 Polynommultiplikation (Divide & Conquer)

Dieser Algorithmus soll rekursiv implementiert werden wobei darauf zu achten ist, dass bereits implementierte Funktionen wiederverwendet werden und dass das Zusammenfügen in einer eigenen Funktion gehandhabt wird, da dieses Zusammensetzen, laut Angabe, nicht gerade trivial ist. Da die Information benötigt wird, wie der originale Grad der Polynome war, soll die Schnittstelle um diese Variable erweitert werden, da innerhalb der rekursiven Aufrufe, diese Information benötigt wird, da ansonsten nicht klar ist, welche Teilhälfte des Polynoms gerade bearbeitet wird.
Verwendete Arrays sollen so effizient wie möglich wiederverwendet werden, dies kann aber die Lesbarkeit des Code negativ beeinflussen.

Vorgehensweise:

1. Berechne die Indexes der Teilhälften (0 ← → low | (low + 1) ←→ grade)
2. Rufe die Funktion rekursiv für die Teilhälften auf.
3. Führe die Berechnungen der Teilhälften laut Angabe durch.
4. Nachdem die Teilaufgaben gelöst wurden füge die Teileergebnisse laut Angabe zusammen

Das Resultat soll sich mit der bereits implementierten Funktion decken, sofern die beiden korrekt funktionieren.

Einschränkungen:

1. Die laut Angaben

3.2 Implementierung

Folgend ist der Source die Header Datei und die Makefile des Programms polynom angeführt.

3.2.1 polynom.h

```
/*
 * ploynone.h
 *
 * This header file declares the functions used for calculating polynoms.
 * For further details see function doc.
 *
 *  Created on: Oct 1, 2014
 *      Author: Thomas Herzog
 */


#ifndef PLOYNONE_H
#define PLOYNONE_H_

#include "common.h"

/**
 * Prints a ploynome in the format '1 + 1x + 1x^2 + ... + nx^n' depending on the given array.
 *
 * int n: the count of elements in the array. last index n - 1
 * double p[]: the array representing the Ploynom where the index represents the power and the
value the base.
 */
void PrintPoly(int n, double p[]);

/**
 * Evaluates a polynom via the horner shema and calculates the
 *
 * int x: value for x within the ploynom
 * int n: the count for ploynom parts
 * double p[]: the array representing the polynom
 */
double EvalPoly(int x, int n, double p[]);

/**
 * Calculates the summary of two given polynoms
 *
 * int np: the count of the first polynom parts
 * double p[]: the array representing the first polynom
 * int nq: the count of the second polynom parts
 * double q[]: the array representing the second polynom
 * double r[]: the representing calculated sum of the two polynoms
 */
int PolySum(int np, double p[], int nq, double q[], double (*r)[]);

/**
 * Calculates the product of the two given polynoms.
 *
 * int np: the count of the first polynom parts
 * double p[]: the array representing the first polynom
 * int nq: the count of the second polynom parts
 * double q[]: the array representing the second polynom
 * double r[]: the representing calculated product of the two polynoms
 */
int PolyProd(int np, double p[], int nq, double q[], double (*r)[]);

/**
 * Calculates the product of the two given polynoms via an divide & conquer paradigm.
 *
 * int np: the count of the first polynom parts
 * double p[]: the array representing the first polynom
 * int nq: the count of the second polynom parts
 * double q[]: the array representing the second polynom
 * double r[]: the representing calculated product of the two polynoms
 */
int PolyProdFast(int np, double p[], int nq, double q[], double (*r)[]);

/**
 * Splits the ploynom between the low and high border.
```

```
 * int low: the low border of the polynom
 * int high: the hig h border of an polynom
 * double p[]: the polynom to split
 *  double (*r)[]: the split ploynom
 */
int PolySplit(int low, int high, double p[], double (*r)[]);

/**
 * This function calculates the product between an polynom part (E.g.: 4x^3)
 * with an complete polynom (E.g.: 1 + 2x + 5x^2). The calculated product
 * is an new polynom.
 *
 * int grade: the grade of the polynom part
 * double polyPart: the coefficient for he polynom part
 * int n: the count of elements of the array representing the polynom to be multiplied by the
polynom part
 * double p[]: the array representing the polynom to be multiplied by the polynom part
 * double (*r)[]: the array which will hold the calculated polynom product
 */
int PolyMultiply(int grade, double polyPart, int n, double p[], double (*r)[]);

#endif /* PLOYNONE_H_ */
```

3.2.2 polynom.c

```
/*
 * polynome.c
 *
 * This program handles polynoms with different algorithms.
 *
 *  Created on: Oct 1, 2014
 *      Author: Thomas Herzog
 */
#include "polynom.h"
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <math.h>

int main(int argc, char** argv) {
    double r1[100], r2[100], r3[100];
    double result;
    int n;

    /* The following tests the PrintPoly function */
    printf("-------------------------------------------\n");
    printf("Tests for the printPoly function\n");
    printf("-------------------------------------------\n");

    printf("\nCase 1: { 0, 0, 0 }\n");
    r1[0] = 0.0;
    r1[1] = 0.0;
    r1[2] = 0.0;
    printf("p = ");
    PrintPoly(2, r1);

    printf("\nCase 2: { 1, 0, 0 }\n");
    r1[0] = 1.0;
    r1[1] = 0.0;
    r1[2] = 0.0;
    printf("p = ");
    PrintPoly(2, r1);

    printf("\nCase 3: { 0, 1, 0 }\n");
    r1[0] = 0.0;
    r1[1] = 1.0;
    r1[2] = 0.0;
    printf("p = ");
    PrintPoly(2, r1);

    printf("\nCase 4: { 0, 0, 1 }\n");
    r1[0] = 0.0;
    r1[1] = 0.0;
    r1[2] = 1.0;
```

```
    printf("p = ");
    PrintPoly(2, r1);

    printf("\nCase 5: { -1, 0, 0 }\n");
    r1[0] = -1.0;
    r1[1] = 0.0;
    r1[2] = 0.0;
    printf("p = ");
    PrintPoly(2, r1);

    printf("\nCase 6: { 0, -1, 0 }\n");
    r1[0] = 0.0;
    r1[1] = -1.0;
    r1[2] = 0.0;
    printf("p = ");
    PrintPoly(2, r1);

    printf("\nCase 7: { 0, 0, -1 }\n");
    r1[0] = 0.0;
    r1[1] = 0.0;
    r1[2] = -1.0;
    printf("p = ");
    PrintPoly(2, r1);

    printf("\nCase 8: { -12, 0.0, -10.65, -1, 0, -89.25 }\n");
    r1[0] = -12;
    r1[1] = 0.0;
    r1[2] = -10.65;
    r1[3] = -1;
    r1[4] = 0;
    r1[5] = -89.25;
    printf("p = ");
    PrintPoly(5, r1);

    printf("\nCase 9: { 12, 0.0, -10.65, -1, 0, -89.25 }\n");
    r1[0] = 12;
    r1[1] = 0.0;
    r1[2] = -10.65;
    r1[3] = -1;
    r1[4] = 0;
    r1[5] = -89.25;
    printf("p = ");
    PrintPoly(5, r1);

    printf("\nCase 10: { 0.0, 0.0, -10.65, -1, 0, -89.25 }\n");
    r1[0] = 0.0;
    r1[1] = 0.0;
    r1[2] = -10.65;
    r1[3] = -1;
    r1[4] = 0;
    r1[5] = -89.25;
    printf("p = ");
    PrintPoly(5, r1);

    printf("\nCase 11: { 0.0, 0.0, -10.65, 0, -1, 0.0 }\n");
    r1[0] = 0.0;
    r1[1] = 0.0;
    r1[2] = -10.65;
    r1[3] = 0;
    r1[4] = -1;
    r1[5] = 0;
    printf("p = ");
    PrintPoly(5, r1);

    printf("\n\n-----------------------------------------------\n");
    printf("Tests for the EvalPoly function\n");
    printf("-----------------------------------------------\n");

    printf("\nCase 1:\n");
    r1[0] = 0;
    r1[1] = 0;
    r1[2] = 0;
    n = 2;
    printf("p(%d) = ", n);
    PrintPoly(2, r1);
    result = EvalPoly(n, 2, r1);
```

```
    printf("Evaluated to %3.2f\n", result);

    printf("\nCase 2:\n");
    r1[0] = -1;
    r1[1] = 0;
    r1[2] = 0;
    n = 2;
    printf("p(%d) = ", n);
    PrintPoly(2, r1);
    result = EvalPoly(n, 2, r1);
    printf("Evaluated to %3.2f\n", result);

    printf("\nCase 3:\n");
    r1[0] = 0;
    r1[1] = -1;
    r1[2] = 0;
    n = 2;
    printf("p(%d) = ", n);
    PrintPoly(2, r1);
    result = EvalPoly(n, 2, r1);
    printf("Evaluated to %3.2f\n", result);

    printf("\nCase 4:\n");
    r1[0] = 0;
    r1[1] = 0;
    r1[2] = -1;
    n = 2;
    printf("p(%d) = ", n);
    PrintPoly(2, r1);
    result = EvalPoly(n, 2, r1);
    printf("Evaluated to %3.2f\n", result);

    printf("\nCase 5:\n");
    r1[0] = 11;
    r1[1] = 7;
    r1[2] = -5;
    r1[3] = -4;
    r1[4] = 2;
    n = 2;
    printf("p(%d) = ", n);
    PrintPoly(4, r1);
    result = EvalPoly(n, 4, r1);
    printf("Evaluated to %3.2f\n", result);

    printf("\n\n-----------------------------------------------\n");
    printf("Tests for the PolySum function\n");
    printf("-----------------------------------------------\n");

    printf("\nCase 1: \n");
    r1[0] = 0;
    r1[1] = 0;
    printf("p = ");
    PrintPoly(1, r1);
    r2[0] = 0;
    r2[1] = 0;
    printf("q = ");
    PrintPoly(1, r2);
    printf("p + q = ");
    PrintPoly((PolySum(1, r1, 1, r2, &r3)), r3);

    printf("\nCase 2: \n");
    r1[0] = 0;
    r1[1] = 0;
    printf("p = ");
    PrintPoly(1, r1);
    r2[0] = 1;
    r2[1] = 2;
    r2[2] = -5;
    r2[3] = -3;
    printf("q = ");
    PrintPoly(3, r2);
    printf("p + q = ");
    PrintPoly((PolySum(1, r1, 3, r2, &r3)), r3);

    printf("\nCase 3: \n");
    r1[0] = 1;
```

```
    r1[1] = 2;
    r1[2] = 3;
    r1[3] = -4;
    printf("p = ");
    PrintPoly(3, r1);
    r2[0] = 0;
    r2[1] = 0;
    printf("q = ");
    PrintPoly(1, r2);
    printf("p + q = ");
    PrintPoly((PolySum(3, r1, 1, r2, &r3)), r3);

    printf("\nCase 4: \n");
    r1[0] = 1;
    r1[1] = 2;
    r1[2] = 3;
    r1[3] = -4;
    printf("p = ");
    PrintPoly(3, r1);
    r2[0] = 1;
    r2[1] = 2;
    r2[2] = -5;
    r2[3] = -3;
    printf("q = ");
    PrintPoly(3, r2);
    printf("p + q = ");
    PrintPoly((PolySum(3, r1, 3, r2, &r3)), r3);

    printf("\nCase 5: \n");
    r1[0] = 1;
    r1[1] = 1;
    r1[2] = 3;
    r1[3] = -4;
    r1[4] = 6;
    printf("p = ");
    PrintPoly(4, r1);
    r2[0] = 1;
    r2[1] = 6;
    r2[2] = 6;
    r2[3] = -3;
    r2[4] = 2;
    printf("q = ");
    PrintPoly(4, r2);
    printf("p + q = ");
    PrintPoly((PolySum(4, r1, 4, r2, &r3)), r3);

    printf("\n\n-----------------------------------------------\n");
    printf("Tests for the PolyProd function\n");
    printf("-----------------------------------------------\n");
    printf("\nCase 1: \n");
    r1[0] = 0;
    r1[1] = 0;
    printf("p = ");
    PrintPoly(1, r1);
    r2[0] = 0;
    r2[1] = 0;
    printf("q = ");
    PrintPoly(1, r2);
    printf("p * q = ");
    PrintPoly((PolyProd(1, r1, 1, r2, &r3)), r3);

    printf("\nCase 2: \n");
    r1[0] = 0;
    r1[1] = 0;
    printf("p = ");
    PrintPoly(1, r1);
    r2[0] = 1;
    r2[1] = 2;
    r2[2] = -5;
    r2[3] = -3;
    printf("q = ");
    PrintPoly(3, r2);
    printf("p * q = ");
    PrintPoly((PolyProd(1, r1, 3, r2, &r3)), r3);

    printf("\nCase 3: \n");
```

```
    r1[0] = 1;
    r1[1] = 2;
    r1[2] = 3;
    r1[3] = -4;
    printf("p = ");
    PrintPoly(3, r1);
    r2[0] = 0;
    r2[1] = 0;
    printf("q = ");
    PrintPoly(1, r2);
    printf("p * q = ");
    PrintPoly((PolyProd(3, r1, 1, r2, &r3)), r3);

    printf("\nCase 4: \n");
    r1[0] = 1;
    r1[1] = 2;
    r1[2] = 3;
    r1[3] = -4;
    printf("p = ");
    PrintPoly(3, r1);
    r2[0] = 1;
    r2[1] = 2;
    r2[2] = -5;
    r2[3] = -3;
    printf("q = ");
    PrintPoly(3, r2);
    printf("p * q = ");
    PrintPoly((PolyProd(3, r1, 3, r2, &r3)), r3);

    printf("\nCase 5: \n");
    r1[0] = 1;
    r1[1] = 1;
    r1[2] = 6;
    r1[3] = -4;
    printf("p = ");
    PrintPoly(3, r1);
    r2[0] = 1;
    r2[1] = 6;
    r2[2] = 6;
    r2[3] = -3;
    printf("q = ");
    PrintPoly(3, r2);
    printf("p + q = ");
    PrintPoly((PolyProd(3, r1, 3, r2, &r3)), r3);

    printf("\nCase 6: \n");
    r1[0] = 1;
    r1[1] = 1;
    r1[2] = 3;
    r1[3] = -4;
    printf("p = ");
    PrintPoly(3, r1);
    r2[0] = 1;
    r2[1] = 2;
    r2[2] = -5;
    r2[3] = -3;
    printf("q = ");
    PrintPoly(3, r2);
    printf("p + q = ");
    PrintPoly((PolyProd(3, r1, 3, r2, &r3)), r3);

    printf("\nCase 7: \n");
    double p0[] = { 7, 0, -4, 3, 1, 0, -1, 3, 7, 0, -4, 3, 1, 0, -1, 3 };
    double q0[] = { -1, 3, -2, 3, 2, -1, 1, -2, 7, 0, -4, 3, 1, 0, -1, 3 };
    printf("p = ");
    PrintPoly(15, p0);
    printf("q = ");
    PrintPoly(15, q0);
    printf("p + q = ");
    PrintPoly((PolyProdFast(15, p0, 15, q0, &r3)), r3);

    printf("\n\n-------------------------------------------\n");
    printf("Tests for the PolyProdFast function\n");
    printf("-------------------------------------------\n");

    printf("\nCase 1: \n");
```

```
    r1[0] = 1;
    r1[1] = 2;
    r1[2] = 3;
    r1[3] = -4;
    printf("p = ");
    PrintPoly(3, r1);
    r2[0] = 1;
    r2[1] = 2;
    r2[2] = -5;
    r2[3] = -3;
    printf("q = ");
    PrintPoly(3, r2);
    printf("p * q = ");
    PrintPoly((PolyProdFast(3, r1, 3, r2, &r3)), r3);

    printf("\nCase 2: \n");
    r1[0] = 1;
    r1[1] = 1;
    r1[2] = 6;
    r1[3] = -4;
    printf("p = ");
    PrintPoly(3, r1);
    r2[0] = 1;
    r2[1] = 6;
    r2[2] = 6;
    r2[3] = -3;
    printf("q = ");
    PrintPoly(3, r2);
    printf("p + q = ");
    PrintPoly((PolyProdFast(3, r1, 3, r2, &r3)), r3);

    printf("\nCase 3: \n");
    r1[0] = 1;
    r1[1] = 1;
    r1[2] = 3;
    r1[3] = -4;
    printf("p = ");
    PrintPoly(3, r1);
    r2[0] = 1;
    r2[1] = 2;
    r2[2] = -5;
    r2[3] = -3;
    printf("q = ");
    PrintPoly(3, r2);
    printf("p + q = ");
    PrintPoly((PolyProdFast(3, r1, 3, r2, &r3)), r3);

    printf("\nCase 4: \n");
    double p[] = { 7, 0, -4, 3, 1, 0, -1, 3, 7, 0, -4, 3, 1, 0, -1, 3 };
    double q[] = { -1, 3, -2, 3, 2, -1, 1, -2, 7, 0, -4, 3, 1, 0, -1, 3 };
    printf("p = ");
    PrintPoly(15, p);
    printf("q = ");
    PrintPoly(15, q);
    printf("p + q = ");
    PrintPoly((PolyProdFast(15, p, 15, q, &r3)), r3);

    return EXIT_SUCCESS;

}

void PrintPoly(int n, double p[]) {
    int i;
    double coefficient;
    bool firstPrinted = false;

    for (i = 0; i <= n; ++i) {
        coefficient = p[i];

        /* Need to print part */
        if (coefficient != 0) {
            /* Add correct algebraic sign */
            if ((firstPrinted)) {
                if (coefficient < 0) {
                    printf(" - ");
                } else if (firstPrinted) {
```

```
                    printf(" + ");
                } /* if */
            } /* if */
            /* Print coefficient */
            if ((i == 0) || (fabs(coefficient) != 1)) {
                printf("%g", (!firstPrinted) ? coefficient : fabs(coefficient)); /* only need
to consider algebraic sign on first coefficient a0 */
                firstPrinted = true;
            } /* if */
            /* Add exponent */
            if (i > 0) {
                if ((i > 0) && (!firstPrinted) && (coefficient == -1)) {
                    printf("-");
                }
                printf("x");
                if (i > 1) {
                    printf("^%d", i);
                } /* if */
                firstPrinted = true;
            } /* if */
        } /* if */
    } /* for */

    printf("\n");
}

double EvalPoly(int x, int n, double p[]) {
    int i;
    double result = 0;
    bool firstUsed = false;

    if (n > 0) {
        for (i = (n); i > 0; --i) {
            /* Set result to first coefficient */
            if (!firstUsed) {
                result = p[i];
                firstUsed = true;
            } /* if */
            /* calculate via horner schema */
            result *= x;
            result += p[i - 1];
        } /* for */
    } /* if */

    return result;
}

int PolySum(int np, double p[], int nq, double q[], double (*r)[]) {
    int i, pval, qval, border, grade;

    /* Even result is 0 still valid grade */
    grade = -1;
    /* Border is array with highest grade */
    border = ((np > nq) ? np : nq);
    for (i = 0; i <= border; ++i) {
        /* consider array borders */
        pval = (i <= np) ? p[i] : 0;
        qval = (i <= nq) ? q[i] : 0;
        /* set sum on result */
        (*r)[i] = pval + qval;
        /* Increase grade if no a0 and value on index i != 0 */
        if ((*r)[i] != 0) {
            grade = i;
        } /* if */
    } /* for */

    return grade;
}

int PolyProd(int np, double p[], int nq, double q[], double (*r)[]) {
    int i, rGrade, r1Grade;
    double r1[100];

    rGrade = -1;
    for (i = 0; i <= nq; ++i) {
        r1Grade = PolyMultiply(i, q[i], np, p, &r1);
```

```
        rGrade = PolySum(r1Grade, r1, rGrade, *r, r);
    } /* for */

    return rGrade;
}

int PolyProdFast(int np, double p[], int nq, double q[], double (*r)[]) {
    int i, half, grml, grmh, grm, grl, grh, n, gSum;
    double rml[100], rmh[100], rm[100], rl[100], rh[100], rlInv[100],
           rhInv[100], rSum[100];
    /* np and nq are grade */
    n = (np + 1);
    half = (n / 2) - 1;

    grml = PolySum(half, p, half, (p + half + 1), &rml);
    grmh = PolySum(half, q, half, (q + half + 1), &rmh);

    if (n > 4) {
        grl = PolyProdFast(half, p, half, q, &rl);
        grh = PolyProdFast(half, (p + half + 1), half, (q + half + 1), &rh);
        grm = PolyProdFast(grml, rml, grmh, rmh, &rm);
    } else {
        grl = PolyProd(half, p, half, q, &rl);
        grh = PolyProd(half, (p + half + 1), half, (q + half + 1), &rh);
        grm = PolyProd(grml, rml, grmh, rmh, &rm);
    } /* if */

    PolyMultiply(0, -1, grl, rl, &rlInv);
    PolyMultiply(0, -1, grh, rh, &rhInv);
    gSum = PolySum(grl, rlInv, grh, rhInv, &rSum);
    gSum = PolySum(gSum, rSum, grm, rm, &rSum);

    /* Init the result */
    for (i = 0; i <= (n * 2 - 1); i++) {
        (*r)[i] = 0;
    } /* for */

    /* Set the result on the proper index */
    for (i = 0; i <= grl; i++) {
        (*r)[i] = rl[i];
    } /* for */
    for (i = 0; i <= gSum; i++) {
        (*r)[n / 2 + i] += rSum[i];
    } /* for */
    for (i = 0; i <= grh; i++) {
        (*r)[n + i] += rh[i];
    } /* for */
    return n * 2 - 1;
}

int PolySplit(int low, int high, double p[], double (*r)[]) {
    int i, j;
    j = -1;
    for (i = low; i <= high; ++i) {
        j++;
        (*r)[j] = p[i];
    } /* for */
    return j;
}

int PolyMultiply(int grade, double polyPart, int n, double p[], double (*r)[]) {
    int rOldGrade, rGrade, i, j;
    rGrade = 0;
    rOldGrade = -1;

    if (n > 0) {
        for (i = 0; i <= n; ++i) {
            rGrade = i + grade;
            j = rOldGrade + 1;
            /* Need to fill space between parts with 0
             * E.g.: x^4 + 4x^5
             * Therefore -> 0x^0, 0x^1, 0x^2, 0x^3 need to be set to 0
             */
            while (j < rGrade) {
                (*r)[j] = 0;
                ++j;
```

```
            } /* while */
            rOldGrade = rGrade;
            (*r)[rGrade] = p[i] * polyPart;
        } /* for */
    } /* if */

    return rGrade;
}
```

3.2.3 Makefile

Sie Programm weight. Da keine besonderen Funktionalitäten vorhanden sind, wird dieselbe makefile verwendet.

3.3 Tests

Folgend sind die Tests des Programms polynom angeführt, welche die verlangten Funktionen testen.

3.3.1 PrintPoly

Folgend sind die Tests für die Funktion PrintPoly angeführt, welches ein Polynom repräsentiert durch ein Array auf der Konsole ausgibt.

```
-------------------------------------------
Tests for the printPoly function
-------------------------------------------

Case 1: { 0, 0, 0 }
p =

Case 2: { 1, 0, 0 }
p = 1

Case 3: { 0, 1, 0 }
p = x

Case 4: { 0, 0, 1 }
p = x^2

Case 5: { -1, 0, 0 }
p = -1

Case 6: { 0, -1, 0 }
p = -x

Case 7: { 0, 0, -1 }
p = -x^2

Case 8: { -12, 0.0, -10.65, -1, 0, -89.25 }
p = -12 - 10.65x^2 - x^3 - 89.25x^5

Case 9: { 12, 0.0, -10.65, -1, 0, -89.25 }
p = 12 - 10.65x^2 - x^3 - 89.25x^5

Case 10: { 0.0, 0.0, -10.65, -1, 0, -89.25 }
p = -10.65x^2 - x^3 - 89.25x^5

Case 11: { 0.0, 0.0, -10.65, 0, -1, 0.0 }
p = -10.65x^2 - x^4
```

Hierbei ist zu sehen, dass der erste Polynomwert egal welchen grades und welches Vorzeichen es hat korrekt erkannt und ausgegeben wird. Bei einem Polynom, welches aus lauter 0 besteht wird nichts ausgegeben.

3.3.2 EvalPoly

Folgend sind die Tests für die Funktion EvalPoly angeführt, welche ein Polynom nach x mit dem Horner Schema berechnet.

```
-------------------------------------------
Tests for the EvalPoly function
-------------------------------------------

Case 1:
p(2) =
Evaluated to 0.00

Case 2:
p(2) = -1
Evaluated to -1.00

Case 3:
p(2) = -x
Evaluated to -2.00

Case 4:
p(2) = -x^2
Evaluated to -4.00

Case 5:
p(2) = 11 + 7x - 5x^2 - 4x^3 + 2x^4
Evaluated to 5.00
```

Sollte kein Polynom angegeben werde, so wird 0 geliefert.

Ansonsten wird das Polynom mit dem Horner Schema korrekt berechnet.

3.3.3 PolySum

Folgend sind die Tests für die Funktion PolySum angeführt, welche ein zwei Polynome aufsummiert.

```
-------------------------------------------
Tests for the PolySum function
-------------------------------------------

Case 1:
p =
q =
p + q =

Case 2:
p =
q = 1 + 2x - 5x^2 - 3x^3
p + q = 1 + 2x - 5x^2 - 3x^3

Case 3:
p = 1 + 2x + 3x^2 - 4x^3
q =
p + q = 1 + 2x + 3x^2 - 4x^3

Case 4:
p = 1 + 2x + 3x^2 - 4x^3
q = 1 + 2x - 5x^2 - 3x^3
p + q = 2 + 4x - 2x^2 - 7x^3

Case 5:
p = 1 + x + 3x^2 - 4x^3 + 6x^4
q = 1 + 6x + 6x^2 - 3x^3 + 2x^4
p + q = 2 + 7x + 9x^2 - 7x^3 + 8x^4
```

Sollten beide oder eines der Polynome leer sein, so wird dies korrekt erkannt und behandelt.

Ansonsten werden die Summen der beiden Polynome korrekt ermittelt.

3.3.4 PolyProd

Folgend sind die Tests für die Funktion PolyProd angeführt, welche ein zwei Polynome miteinander multipliziert.

Sollten beide oder eines der beiden Polynome leer sein so wird jedesmal 0, also kein Ergebnis zurückgeliefert, da hier eine Multiplikation mit 0 stattfindet.
Ansonsten wird das Produkt korrekt berechnet.

3.3.5 PolyProdFast

Folgend sind die Tests für die Funktion PolyProd angeführt, welche ein zwei Polynome miteinander multipliziert und hierbei das Divide & Conquer Prinzip anwendet.

Da wird davon ausgehen konnte, das die Polynome gleichen Grad besitzen und Polynome mit lauter 0 keine Grad besitzen aber es über den Index des Array produziert wurde, ist zu sehen das die Funktion keine korrekten Ergebnisse liefert, was an der Art der Berechnung liegt.
Die in PolyProd getesteten Polynome Multiplikationen liefern jedoch dieselben Ergebnisse.