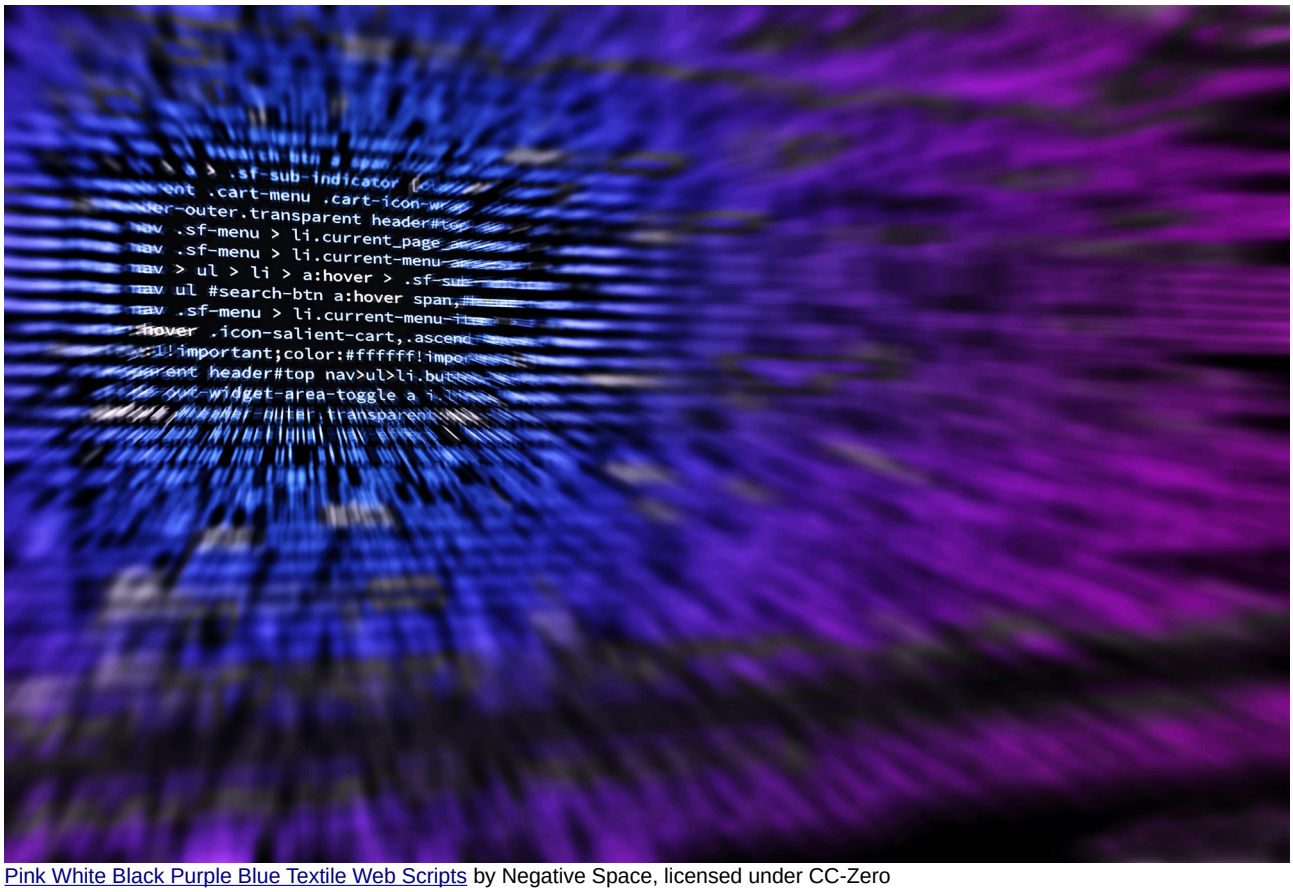
Workbook für das Praktikum Algorithmen & Programmierung 1

Von: Laskar Alexander Theuer
Matrikelnummer: 11378947



Pink White Black Purple Blue Textile Web Scripts by Negative Space, licensed under CC-Zero

Table of Contents

1. Fachfragenkatalog

1.1. Was ist der Unterschied zwischen `Float` und `Double`?

`Float` und `Double` werden beide genutzt, um Fließkommazahlen zu speichern. Der Unterschied liegt darin, dass `Double`, wie der Name schon sagt, doppelte Genauikeit hat. Bei modernen Systemen hat ein `float` üblicherweise 32 bit, ein `double` `64` bit.

1.2. Wofür werden `structs` verwendet, und welche Bestandteile werden benötigt?

`Structs` werden verwendet, um zusammenhängende Daten an einem Ort zu speichern. Genauer gesagt wird der Code im Computer dadurch in direkt nebeneinander liegenden Teilen des Arbeitsspeichers gespeichert. Structs bestehen immer aus dem Keyword `struct`, gefolgt von den Namen, einer geöffneten geschweiften Klammer `{`, Inhalten des struct, getrennt durch Kommata, einer schließenden geschweiften Klammer `}`, abgeschlossen mit einem Semikolon. `Structs` selbst können, anders als Klassen, keine Methoden oder Funktionen enthalten, wohl aber function pointers. Hier ein Beispiel:

```c
#include <stdbool.h> // Für den bool typedef, sowie true und false

struct Person {
    char* name; // Hier ein char Pointer, um strings darzustellen, da die größe zum Zeitpunkt der Definition noch nicht bekannt ist.
    int alter; // Ganze Zahl, die das Alter der Person repräsentiert
    bool lebendig; // Bool, der angibt, ob die Person lebt
    void (*lebe)(); // pointer zu einer lebe funktion
};

void lebe_Person(struct Person* person) { // Implementation der Lebe Funktion aus dem Person struct.
    if (!person->lebendig) { // Führe den Code nur aus, wenn die Person nicht lebt
        person->lebendig = true; // Macht die Person lebendig
    }
}
```

Oft wird auch ein `typedef struct StructName StructName` verwendet, damit man nicht immer struct davor schreiben muss.

1.3. Was ist der Unterschied zwischen Syntax und Semantik?

Syntax bezeichnet die korrekte Formulierung von Code auf einer rein Formellen Basis. Sind alle Semikolons da, wo sie sein sollen? Sind alle (geschweiften) Klammern richtig gesetzt? Wurden korrekte Typen angegeben? Diese Dinge werden überprüft während das Programm kompiliert wird. Bei Fehlern wird der Compiler die Kompilation beenden und einen Fehler ausgeben. Semantik hingegen bezeichnet die Logik, die das Programm ausführt. Werden die richtigen Konditionen überprüft? Werden Variablen oder Konstanten die richtigen Werte zugewiesen? Werden die

richtigen Funktionen aufgerufen? Beispiel für ein Programm mit einem Syntaxfehlern:

```c
#include <stdio.h>

void main() { //Deklaration von main als void funktion ist falsch
    printf("Hello Word!\n") // Fehldes Semikolon
    return 0; // Void Funktion gibt einen Wert aus.
}
```

Beispiel für ein Programm mit einem Semantikfehler:

```c
#include <ctype.h> // Inkludiert für isdigit()
#include <stdbool.h> // Inkludiert für den Bool typedef
#include <stdio.h> // Für printf

bool is_int(char* str) { // Funktion, die überprüft, ob ein gegebener String
eine ganze Zahl ist oder nicht
    if (str == NULL) {return false;} // Funktion gibt falsch zurück, wenn der
gegebene String NULL ist. Sicherheitsmaßnahme um zu verhindern, dass ein NULL
pointer dereferenziert wird.
    while (str != '\0') { // Code wird ausgeführt, bis der NULL Terminator
erreicht wird
        if (!isdigit(str) && !(str == '0')) { // Überprüft, ob der aktuelle
char entweder nicht 1-9 oder nicht 0 ist.
            return true; // Die Funktion gibt hier true zurück, obwohl ein
Zeichen außer 0-9 vorkommt.
        }
        str++; // Geht zum nächsten Character in str.
    }
    return false; // Gibt false zurück, obwohl es eigentlich ein Integer ist.
}

// Zum verdeutlichen:
int main() {
    char[10] num = "0123456789"; // Definitiv eine ganze Zahl
    if (is_int(&num)) {
        printf("Es ist eine Zahl.\n");
    } else {
    printf("Es ist keine Zahl.\n");
    }
    return 0;
}
```

Dieser Code würde `Es ist keine Zahl.` ausgeben, obwohl es definitiv eine Zahl ist. Die meisten bugs werden durch Semantikfehler verursacht, selten sind diese auch in der Standardbibliothek vorhanden. Durch Semantikfehler entstehen auch größere Probleme, wie z. B. NULL-ptr dereference, free-after-use etc.

1.4. Was versteht man unter dem `fall-through`? Wie ist diese verhalten zu verhindern, wann könnte es nützlich sein?

`fall-through` beschreibt ein Verhalten bei `switch` statements, wenn nach einem `case` Block kein `break;` kommt und dadurch der nächste Fall bis zu einem `break` ausgeführt wird. Als Beispiel:

```c
#include <stdio.h>
int main() {
    int wert = 1;
```

```
    switch (wert) {
        case 1:
            printf("Der Wert ist 1.\n"); // Kein break; in diesem Block
        case 2:
            printf("Der Wert ist 2.\n");
            break;
        case 3:
            printf("Der Wert ist 3.\n");
            break;
        default:
            printf("Der Wert ist weder 1 noch 2 noch 3.\n"); // Hier braucht es
kein break; weil das der letzte Block ist.
    }
    return 0;
}
```

Hier tritt ein `fall-through` auf, weil in dem Block von `case 1` kein `break;` folgt. Verhindert werden kann dieses Verhalten, indem nach jedem Block ein `break;` geschrieben wird, wodurch das `switch` statement sofort beendet wird. Ein `fall-through` kann aber auch nützlich sein, z. B. wenn man möchte, dass Code ab einem bestimmten Punkt ausgefürt wird. Ein Beispiel:

```
#include <stdio.h>

int berechne_semester_bis_studienende(int bestandene_semester) { // Hier wird
die Regelstudienzeit angenommen
    switch (bestandene_semester) {
        case 0:
            bestande_semester++;
        case 1:
            bestande_semester++;
        case 2:
            bestande_semester++;
        case 3:
            bestande_semester++;
        case 4:
            bestande_semester++;
        case 5:
            bestande_semester++;
    }
    return bestande_semester; // Eigentlich könnte man diese ganze Funktion mit
6-bestandene_semester ersetzen
}

int main() {
    int buffer;
    printf("Gib ein, wie viele Fachsemester du bereits bestanden hast: ");
    scanf("%d", &buffer);
    printf("Dein Studium wäre in Regelstudienzeit in %d Monaten abgeschlossen.
\n", berechne_semester_bis_studienende(buffer));
    return 0;
}
```

Man könnte diesen Code zwar grundsätzlich anders einfacher formulieren, aber das ist ein Beispiel, wo `switch` und `fall-through` als Einstiegspunkt genutzt werden. Oft sind `fall-throughs` unabsichtlich. Ein Entwickler hat ein `break;` vergessen, weshalb sie in solchen Fällen *Semantikfehler* sind. Dies ist aber nicht immer der Fall. `Fall-throughs` sind eine durchaus nützliche Programmierpraxis, wenn der geswitchte Wert als Einstiegspunkt verstanden wird und damit in eine Menge von immer gleich bleibende Operationen auch erst an einem späteren Punkt eingestiegen werden kann, zum Beispiel wenn bestimmte Bedingungen bereits erfüllt sind.

1.5. Erklären Sie den konditionalen (ternären) Operator Anhand eines Beispieles.

Der *ternäre Operator* folgt der Syntax `Bedingung ? ausdruck_wenn_war : ausdruck_wenn_falsch;`. Hier ein Beispiel, wo zugewiesen wird, ob jemand Erwachsen oder Minderjährig ist, abhängig vom Alter.

```c
#include <stdio.h>

int main() {
    int buffer;
    printf("Gib dein Alter ein. ");
    scanf("%d", &buffer);
    char *person = ((buffer < 18) ? "Minderjähriger" : "Erwachsener"); //
Klammern helfen bei der Leserlichkeit. String literals sind eigentlich nur
const char pointers.
    printf("Du bist ein %s.\n", person);
    return 0;
}
```

In diesem Beispiel gibt der Nutzer ein numerisches Alter ein und Mithilfe des ternären Operators wird dem Nutzer entweder Erwachsener oder Minderjähriger zugewisen. Minderjähriger falls Alter < 18 ist, in allen anderen Fällen Erwachsener.

1.6. Was ist der Unterschied zwischen Pre- und Postinkrementation und wie werden sie notiert?

Beide nutzen den *Inkrementationsoperator*, das `++`. Bei der Preinkrementation wird Die Variable sofort inkrementiert, und der alte Wert kann nicht noch anderswo zwischengespeichert werden. Pre, also vor, steht dabei für *vor dem zuweisen*. Sie wird mit `++a` notiert, wenn a die Variable ist, die inkrementiert werden soll. Bei der Postinkrementation, post steht hier für nach, wird die Variable inkrementiert, **nachdem** der Wert einer anderen Variable zugewiesen wurde, wenn man das machen möchte. Die Notation hierfür ist `a++`, wenn a die Variable ist, die inkrementiert werden soll. Zur Verdeutlichung kann dieser Code helfen:

```c
#include <stdio.h>

int main() {
    int a = 5;
    int b = 5;
    int c = a++;
    int d = ++b;
    printf("A: %d\n", a);
    printf("B: %d\n", b);
    printf("C: %d\n", c);
    printf("D: %d\n", d);
    return 0;
}
```

Die Ausgabe hiervon ist:

```
A: 6
B: 6
C: 5
D: 6
```

Daran kann man das Verhalten gut erkennen.

1.7. Was ist der Unterschied zwischen einer `while` und einer `do-while` Schleife?

Beide Schleifen führen Bedingungen aus, solange eine bestimmte Bedingung erfüllt ist. Der große Unterschied dabei ist, dass bei der while-Schleife, der Code unter Umständen gar nicht ausgeführt wird, wenn die Eingangsbedingung nicht wahr ist. Bei der do-while Schleife wird der Code mindestens einmal ausgeführt. Hier beide Schleifen in einem Code Beispiel:

```c
#include <stdio.h>
#include <stdbool.h>

int main() {
    while (false) { // Wird niemals ausgeführt.
        printf("Ausgabe aus der while-Schleife.\n");
    }

    do { // Wird ausgeführt
        printf("Ausgabe aus der do-while Schleife.\n");
    } while (false); // Ab hier wird nicht mehr ausgeführt
    return 0;
}
```

1.8. Welche Probleme können bei einer rekursiven Implementation einer Funktion auftreten?

Rekursive Implementation können verschiedene Probleme aufweisen. Ein häufiges Problem ist eine extrem lange Laufzeit (große Zeitkomplexität). in gutes Beispiel hierfür ist eine rekursive Berechnung einer Zahl aus der Fibonacci Reihe (ohne caching). Hierbei werden häufig immer wieder die gleichen Zahlen als Zwischenschritt berechnet, weshalb die Laufzeit für die Berechnung vom 100sten Element viele Jahre dauert. Dadurch sind rekursive Ansätze für große Berechnungen meist aus Performance gründen nicht sinnvoll. Ein anderes Problem wäre unendliche Laufzeit, wenn ein bestimmte Bedingung gar nicht erfüllt werden kann, in welchem Fall sich die Funktion selbst immer wieder aufruft und dadurch nie endent. So etwas kann in der Implementation verhindert werden.

1.9. Was ist der Unterschied zwischen einer Call-by-Reference und einer Call-by-Value Funktion? Nennen Sie für beide Arten Beispielfunktionen aus der C-Standardbibliothek.

Bei einer Call-by-Value funktion, wird direkt ein Wert in die Funktion gegeben. Dieser wird dann kopiert und alle Modifikationen dieses Wertes passieren in der Kopie, statt in dem Wert (z. B. einer Variable), die der Funktion direkt gegeben wurde. Eine Call-By-Reference Funktion bekommt stattdesssen einen pointer zu einer Variable. Dadurch das statt einem Wert eine Speicheradresse gegeben wird, wird der Wert der Variable direkt modifiziert. Dementsprechend sind call-by-reference Funktionen genau dann sinnvoll, wenn der Wert der Variable in der Funktion selbst modifiziert werden soll. Ein Besispiel für eine Call-By-Reference Funktion wäre zum Beispiel `scanf()` auf `stdio.h`, welche eine Speicheradresse zu dem Buffer bekommt, in den geschrieben

werden soll. Ein Beispiel für eine Call-By-Value Funktion ist z. B. `strcmp()` aus `string.h`, in der zwei Strings verglichen werden. Da die strings hier selbst nicht bearbeitet werden müssen, sondern nur die Werte verglichen werden, ist dieser Ansatz hier sinnvoller.

1.10. Aus welchem Grund sollte die `goto`-Anweisung bei der Programmierung grundsätzlich vermieden werden?

Code, der über `goto` strukturiert ist, neigt dazu sehr unleserlich, unübersichtlich und schwer verständlich zu werden. Es gibt allerdings Ausnahmefälle, in denen `goto` die Leserlichkeit verbessern kann, diese sind jedoch selten. `goto` Kann außerdem dazu genutzt werden, um die Struktur des C codes ähnlicher zu der von Assembly code zu machen, was in seltenen Niechenfällen nützlich sein kann.

1.11. Wodurch wird ein Character-Array zu einem String?

Strings sind `char` arrays, die mit einem Besondern byte, also `char`, dem sogenannten Null-Terminator (`'\0'`), beendet werden. Dadurch können Funktionen, die mit den strings arbeiten, erkennen, das a) der gegebene Array tatsächlich ein String ist und b) der String am Ende tatsächlich beendet ist.

1.12. Welche Informationen befinden sich in einer Deklarationsdatei (Header)?

Ein Header enthält hauptsächlich leere Implementation von Funktionen (Nur die Kopfzeile wo Name, typ und Parameter spezifiziert werden), `structs`, aber keine Implementationen von structs, d. H. in dem struct werden keine Werte zugeordnet, und außerdem Deklarationen von globalen Variablen (aber nur namentlich, ohne Wertzuweisung), Makros und typedefs. Zudem findet in headern oft das inkludieren von Bibliotheken statt, die dann in dem Programm genutzt werden. Die Funktionen werden dann in Dateien mit dem gleichen Namen (Mit der Dateiendung .c) implementiert. Diese Datei inkludiert den Header. Ferner gibt es in header Dateien oft sogenannte *include guards*, welche dazu dienen, redundaten imports zu vermeiden. Ein Beispiel für eine Header Datei wäre zum Beispiel:

```
// Code aus library.h
#ifndef LIBRARY_H_ // Überprüft, ob LIBRARY_H_ definiert ist. Wenn nicht geht
es weiter, wenn doch wird der include direkt beendet.
#define LIBRARY_H_ // Macro als Teil des Include guards definiert

#include <stdio.h> // Importiert stdio.h, damit es später benutzt werden kann.
#include <stdlib.h>

#define ARR_SIZE 1024 // "Magische Zahl", wird hier benutzt um eine konstante
größe von Arrays festzulegen.
typedef unsigned char uchar; // Definiert den typ uchar, der gleichbedeutend
mit unsigned char ist. Nützlich für 8-bit Zahlen von 0-8

struct Tier {
    int alter;
```

```
    char spezies[ARR_SIZE];
    uchar sprunghöhe;
};

struct Tier* erschaffe_tier(int alter, char* spezies, uchar sprunghöhe);
void springe (struct Tier tier);

int Tieranzahl;


#endif // LIBRARY_H_
```

Das hier wäre die Header Datei. Sie könnte zum Beispiel so benutzt werden:

```
// Code in library.c
#include "library.h"

struct Tier* erschaffe_tier(int alter, char* spezies, uchar sprunghöhe) {
    struct Tier* neues_tier = (struct Tier*) malloc(sizeof(struct tier));
    neues_tier->alter = alter;
    neues_tier->spezies = spezies;
    neues_tier->sprunghöhe = sprunghöhe;
    tieranzahl++;
    return neues_tier;
}

void springe(struct Tier tier) {
    printf("Das tier ist %d meter hoch gesprungen.", tier.sprunghöhe);
}
```

Das könnte dann in einer anderen Datei benutzt werden:

```
// code in main.c
#include "library.h"

int main(){
    struct Tier* graues_riesenkänguru = erschaffe_tier(5, "Känguru", 3);
    springe(*graues_riesenkänguru);
    printf("Das tier ist %d Jahre alt.\n", graues_riesenkänguru->alter);
    printf("Das tier hat die Spezies %s\n", graues_riesenkänguru->spezies);
    printf("Insgesamt wurden %d Tiere erschaffen", tieranzahl);
    return 0;
}
```

1.13. Was ist der Prä-Prozessor, wie wird er angesprochen und welche Aufgaben erledigt er?

Der Präprozessor in C ist ein Teil des Kompilierungsvorgangs jedes C-Programms. Er übernimmt im wesentlichen drei Aufgaben:

1. Macro Substitutionen: Alle Konstanten, die mit `#define` definiert wurden, werden textbasiert ersetzt. So wird zum Beispiel `ENOMOM`, der Errorcode für wenn kein Speicher mehr verfügbar ist, durch die Zahl 12 ersetzt, vorrausgesetzt `errno.h` wurde inkludiert.
2. Datei Inklusion: Alle aussagen die mit `#include` beginnen, werden ausgeführt, bedeutet die Header files werden direkt in den Source Code der Datei kopiert und dann wird weiter kompiliert.

3. Konditionale Kompilation: Statements wie `#ifdef`, `#ifndef`, `#if`, `#else` oder `#endif` ermöglicht Codesegmente, die nur kompiliert werden, wenn bestimmte Bedingungen erfüllt werden. Sie sind zum Beispiel Teil von Include guards oder werden benutzt um bestimmte Konfiguration bei der Kompilierung zu ermöglichen. Der Linux Kernel benutzt in `include/linux/fs.h` in seinem Inode struct z. B.

```
#ifdef CONFIG_SECURITY
    void    *i_security;
#endif
```

Wenn in der Kernel Konfiguration also die erweiterte Sicherheit aktiviert wurde (z. B. als Teil von SELinux), wird damit in das Inode struct dieser Teil mit reinkompiliert, ansonsten nicht.

1. Zeilenkontrolle: `#line`, eine sehr selten genutzte Instruktion, erlaubt es, Zeilenzahl oder Dateinamen in der kompilierten Datei zu ändern. Das kann manchmal für debugging hilfreich sein.

Allgemein sprechen also alle Aussagen die mit # beginnen den Präprozessor an. Dieser macht dann, wie er Name schon sagt, vor der tatsächlichen Kompilierung Änderungen am Code, zum Beispiel werden Makros erweitert oder bestimmte Codesegmente werden entfernt oder hinzugefügt, Quellcode aus anderen Dateien wird in die Datei hinzugefügt oder es werden Anweisungen an den Compiler für debugging an den Compiler weitergegeben.

2. Vorbereitungsaufgaben

2.1. 1. Vorbereitungsaufgabe

```c
#include <stdio.h>

double berechne_gesamtpreis(double preis, double anzahl) {
    return preis*anzahl;
}

int main() {
    int id;
    char name[64];
    double preis;
    int anzahl;
    printf("**********************\n");
    printf("* WWS Produkteingabe *\n");
    printf("**********************\n");
    printf("ID: ");
    scanf("%d", &id);
    printf("Name: ");
    scanf("%s", &name);
    printf("Preis: ");
    scanf("%lf", &preis);
    printf("Anzahl: ");
    scanf("%d", &anzahl);
    printf("========================\n");
    printf("| %d |\n", id);
    printf("| Name: %s |\n", name);
    printf("| Einzelpreis: %lf |\n", preis);
    printf("| Anzahl: %d |\n", anzahl);
    printf("| Gesamtpreis: %lf |\n", berechne_gesamtpreis(preis, anzahl));
    return 0;
}
```

2.2. 2. Vorbereitungsaufgabe

```c
#include <errno.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define ERRORRETURN 10787253
#define ARRSIZE 4096
#define EXITVAL 15326
#define ENOCUSTOMER 692137
#define ENOPRODUCT 691337

struct Product {
    int id;
    char* name;
    double price;
    int amount;
};

struct Customer {
    char* name;
    int id;
};

int find_first_free_customer(struct Customer* array[]) {
    for (int i = 0; i < ARRSIZE; i++) {
        if (array[i] == NULL) {
            return i;
        }
    }
    return -1;
}

int find_first_free_product(struct Product* array[]) {
    for (int i = 0; i < ARRSIZE; i++) {
        if (array[i] == NULL) {
            return i;
        }
    }
    return -1;
}

int find_product_index(int id, struct Product* array[]) {
    for (int i = 0; i < ARRSIZE; i++) {
        if (array[i] != NULL && array[i]->id == id) {
            return i;
        }
    }
    errno = ENOPRODUCT;
    printf("Mit der eingegeben Produktnummer konnte kein Produkt gefunden
werden.\n");
    return ERRORRETURN;
}

int find_customer_index(int id, struct Customer* array[]) {
    for (int i = 0; i < ARRSIZE; i++) {
        if (array[i] != NULL && array[i]->id == id) {
            return i;
```

```
        }
    }
    errno = ENOCUSTOMER;
    printf("Mit der eingegeben Kundennummer konnte kein Kunde gefunden werden.
\n");
    return ERRORRETURN;
}

int add_product(struct Product* products[]) {
    int id, amount;
    char name[100];
    double price;

    printf("**********************\n");
    printf("* WWS Produkteingabe *\n");
    printf("**********************\n");
    printf("Bitte geben Sie eine Produktnummer ein: ");
    scanf("%d", &id);
    printf("Bitte geben Sie einen Produktnamen ein: ");
    scanf("%99s", name);
    printf("Bitte geben Sie einen Preis für das Produkt ein: ");
    scanf("%lf", &price);
    printf("Bitte geben Sie die aktuelle Anzahl im Inventar: ");
    scanf("%d", &amount);

    struct Product* new_product = (struct Product*)malloc(sizeof(struct
Product));
    if (!new_product) {
        printf("Fehler beim Speicher zuweisen.");
        errno = ENOMEM;
        return ERRORRETURN;
    }

    new_product->id = id;
    new_product->name = strdup(name); // Duplicate string to avoid issues.
    new_product->price = price;
    new_product->amount = amount;

    int index = find_first_free_product(products);
    if (index == -1) {
        free(new_product);
        printf("Kein freier Platz für Produkte.");
        errno = ENOMEM;
        return ERRORRETURN;
    }

    products[index] = new_product;
    return 0;
}

int add_customer(struct Customer* customers[]) {
    int id;
    char name[100];

    printf("**********************\n");
    printf("* WWS Kundeneingabe *\n");
    printf("**********************\n");
    printf("Bitte geben Sie eine Kundennummer ein: ");
    scanf("%d", &id);
    printf("Bitte geben Sie einen Kundennamen ein: ");
    scanf("%99s", name);
```

```
    struct Customer* new_customer = (struct Customer*)malloc(sizeof(struct
Customer));
    if (!new_customer) {
        printf("Fehler beim Speicher zuweisen.");
        errno = ENOMEM;
        return ERRORRETURN;
    }

    new_customer->id = id;
    new_customer->name = strdup(name); // Duplicate string to avoid issues.

    int index = find_first_free_customer(customers);
    if (index == -1) {
        free(new_customer);
        printf("Kein freier Platz für Kunden.");
        errno = ENOMEM;
        return ERRORRETURN;
    }

    customers[index] = new_customer;
    return 0;
}

int print_product(struct Product* products[]) {
    int id;

    printf("**********************\n");
    printf("* WWS Produktausgabe *\n");
    printf("**********************\n");
    printf("Bitte geben Sie die gesuchte Produktnummer ein: ");
    scanf("%d", &id);

    int index = find_product_index(id, products);
    if (index == ERRORRETURN) {
        return ERRORRETURN;
    }

    struct Product* product = products[index];
    printf("======================\n");
    printf("Produktnummer: %d\n", product->id);
    printf("Name: %s\n", product->name);
    printf("Einzelpreis: %.2lf\n", product->price);
    printf("Anzahl: %d\n", product->amount);
    printf("Gesamtpreis: %.2lf\n", product->price * product->amount);

    return 0;
}

int print_customer(struct Customer* customers[]) {
    int id;

    printf("**********************\n");
    printf("* WWS Kundenausgabe *\n");
    printf("**********************\n");
    printf("Bitte geben Sie die gesuchte Kundennummer ein: ");
    scanf("%d", &id);

    int index = find_customer_index(id, customers);
    if (index == ERRORRETURN) {
        return ERRORRETURN;
```

```
    }

    struct Customer* customer = customers[index];
    printf("====================\n");
    printf("Kundennummer: %d\n", customer->id);
    printf("Name: %s\n", customer->name);

    return 0;
}

int menu(struct Customer* customers[], struct Product* products[]) {
    int input;

    printf("**********************\n");
    printf("* TH Warenwirtschaft *\n");
    printf("**********************\n");
    printf("Optionen:\n");
    printf("1. Produkteingabe\n");
    printf("2. Kundeneingabe\n");
    printf("3. Produktausgabe\n");
    printf("4. Kundenausgabe\n");
    printf("5. Verlassen\n");
    printf("Eingabe: ");
    scanf("%d", &input);

    errno = 0;

    switch (input) {
        case 1:
            add_product(products);
            if (errno != 0) {return ERRORRETURN;}
            break;
        case 2:
            add_customer(customers);
            if (errno != 0) {return ERRORRETURN;}
            break;
        case 3:
            print_product(products);
            if (errno != 0) {return ERRORRETURN;}
            break;
        case 4:
            print_customer(customers);
            if (errno != 0) {return ERRORRETURN;}
            break;
        case 5:
            return EXITVAL;
        default:
            printf("Ungültige Eingabe. Bitte erneut versuchen.\n");
            break;
    }

    return 0;
}

int main() {
    errno = 0;
    int menuval = 0;
    struct Customer* customers[ARRSIZE] = {NULL};
    struct Product* products[ARRSIZE] = {NULL};

    while (menuval == 0) {
```

```
        menuval = menu(customers, products);
    }

    return errno;
}
```

2.3. 3. Vorbereitungsaufgabe:

2.3.1. Call-By-Reference

Call-By-Reference wird immer dann verwendet, wenn Variablen direkt berbeitet werden müssen, zum Beispiel dann, wenn direkt in ein Array geschrieben wird. An einigen anderen Stellen wird Call-By-Reference auch zur Effizienz benutzt, um Dinge wie die Struct arrays nicht neu in den Speicher kopieren zu müssen (ähnlich wie Java diese Dinge handhabt).

2.3.2. Call-By-Value

Call-By-Value wird dann benutzt, wenn die Werte von Variablen nur gelesen werden müssen.

Hier kurz zu allen verwendeten Funktionen, ob diese Call-By-Value oder Call-By-Reference verwenden, und warum:

2.3.3. Funktionsbeispiele

`printf` verwendet Call-By-Value, weil nur einzelne Variablen kopiert und nur zur Ausgabe gelesen werden müssen.

`scanf` verwendet Call-By-Reference, weil direkt in die Speicheradressen der jeweiligen Variable geschrieben wird.

`malloc` verwendet Call-By-Value. Das Argument ist nur eine Zahl, die angibt wie viel Speicher zugewiesen werden soll. Dafür wird ein Pointer zur Speicheradresse zurückgegeben.

`sizeof` verwendet ebenfalls Call-By-Value. Hier wird ein Datentyp und keine Variable selbst angegeben.

`free` ist eine Call-By-Reference Funktion. Der Speicherplatz an der Speicheradresse der gegebenen Variable wird freigegeben, von daher ist nur die Speicheradresse wichtig, der Wert der Variable uninteressant.

`find_first_free_customer` Benutzt für bessere Effizienz beim Aufruf Call-By-Referene.

`find-first-free_product` Benutzt für bessere Effizienz beim Aufruf Call-By-Referene.

`find_product_index` Benutzt Call-By-Value für die ID, da hier nur der Wert der Variable wichtig ist, und Call-By-Reference für den Array, weil das effizienter ist.

`find_customer_index` Siehe oben.

`add_product` verwendet Call-By-Reference, da direkt in das Array geschrieben werden muss. `add customer` siehe oben.

`print_product` verwendet Call-By-Reference, da innerhalb der Funktion Call-By-Reference Aufrufe zum gleichen Array passieren, hier ist die Begründung auch wieder Effizienz.

`print_customer` siehe oben. `menu` Benutzt Call-By-Reference, da aus der Funktion diverse Call-By-Reference Aufrufe gemacht werden müssen, und Dinge wie das Schreiben in Arrays für die gesamte Laufzeit des Programs bestehen müssen. `main` hat keine Parameter.