

## Mikrocontroller, C-Programmierung

# Selbststudium und Übungen SW4

Martin Vogel, Peter Sollberger V7.3

---

Sie kennen das Konzept von Zeiger und Vektoren und können diese entsprechend anwenden.

## 1. Selbststudium

Gehen Sie nochmals die Folien durch.

## 2. Übungen und Aufgaben

### 1. Aufgabe Taschenrechner polnische Notation:

- a) Es soll das Programm des **Taschenrechners** mit polnischer Notation umgesetzt werden (gemäss Beispiel im Buch K&R S. 74 bis S.77 – siehe Anhang).  
Folgende zusätzliche Informationen hierzu:
  - o Verwenden Sie das Template main.c aus dem Anhang als Vorlage
  - o "stdin" ist hier das File "inputfile1.txt" in dem Pfad von „main“ (File mit netbeans als Editor verändern – Win-Text-Editor macht kein EOF wie gewünscht)
  - o Sie können auch über die Konsole die Eingabe machen – EOF mittels ctrl+del
  - o Alle Funktionen sind zuerst in einem Source-File („main...“ File) zu halten
  - o Zeitbedarf 45': codieren, debugging und testen
- b) Teilen Sie das obige **Taschenrechner** Programm in Quelldatei, Headerfiles und main auf, so dass Sie eine Aufteilung gemäss Buch S.80 (siehe Anhang) erreichen. Speichern Sie Ihr Projekt unter einem neuen Namen und testen Sie Ihr Programm entsprechend.
- c) *Optional:* Erweitern Sie Ihren Taschenrechner mit den Funktionen sin, exp, power

### 2. Aufgabe 4-14 aus dem Buch K&R Seite 88:

Schreiben Sie Makros:

- **MIN (x, y)** liefert den kleineren der beiden Werte
- **MAX (x, y)** liefert den grösseren der beiden Werte
- **SWAP (t, x, y)** vertauscht die zwei Argumente x und y vom Typ t.
- **DEBUGPRINT (\_fmt, ...)** Ausgabe von Debug-Informationen

Testen Sie Ihr Programm mit verschiedenen Testfällen.

### 3. Ampel:

Schreiben Sie eine kleine Ampel-Simulation, welche der Reihe nach die Zustände RED – GREEN – GREEN\_BLINKING – YELLOW – RED – etc. annimmt.

Teilen Sie das Programm in drei Files main.c, ampel.c und ampel.h auf und halten Sie den Zustand als statische, lokale Variable im File ampel.c fest. Das File ampel.c soll die drei Funktionen getState(), nextState() und printState() zur Verfügung stellen, welche Sie im main.c testen. Verwenden Sie für Wartezeiten die Unix-Bibliotheksfunktion unsigned int **sleep**(unsigned int seconds) und #include <unistd.h>

**4. Message Queue:**

Machen Sie eine einfache „Message Queue“, welche mind. die drei Funktionen *initialize*, *enqueue* and *dequeue* enthält. Die „Messages“ sollen aus einem Integerwert bestehen. *enqueue* schreibt die neue „Message“ an das Ende der „Queue“, *dequeue* liest das nächste Element aus der „Queue“. Falls keine „Messages“ vorhanden sind soll die Funktion *dequeue* blockieren. (Tipp: siehe verkettete Liste)

Achten Sie auf ein sauberes „Allokieren und Freigeben“ der Speicherplätze und testen Sie Ihr Programm entsprechend!

Anhang zu Aufgabe 1a – K&R Seite 74ff  
Datei main.c für Taschenrechner

```
#include <stdio.h>
#include <stdlib.h>      /* fuer atof() */

/* hier alle Konstanten */
#define MAXOP    100      /* max Laenge von Operand und/oder Operator */
/* weitere hier anfuegen */

int getop(char []);
/* weitere hier anfuegen */

char buf[BUFSIZE];
/* weitere hier anfuegen */

/* Taschenrechner in umgekehrter polnischer Notation - Beispiel S. 74 K&R */
int main() {
/* hier lokale Variablen definieren */
    freopen("inputfile1.txt", "r", stdin);
    //oeffnet das File "inputfile.txt" in dem Pfad von main.c und nimmt das File als
    //Standard Eingabe anstelle Tastatur
    while ((type = getop(s)) != EOF) {
/* hier ausprogrammieren */
        fclose(stdin);      // Schliesst die Standard-Eingabe stdin (das File von freopen)
        return(0);

/* push: f auf den Stack bringen */
void push(double f) {
/* hier ausprogrammieren */

/*pop: Wert von Stack holen und liefern */
double pop(void) {
/* hier ausprogrammieren */

/*getop: naechsten Operator und /oder numerischen Operanden holen */
int getop(char s[]){
/* hier ausprogrammieren */

/* Zeichen holen mit Buffer */
int getch(void) {
/* hier ausprogrammieren */

/* Zeichen wieder zurueckstellen wenn zuviel geholt */
void ungetch(int c){
/* hier ausprogrammieren */
```

Anhang zu Aufgabe 1a Taschenrechner aus Buch K&R Seite 73ff.

Wir wollen dies mit einem größeren Beispiel vertiefen. Wir beschreiben dazu einen Taschenrechner, der die Operatoren +, -, \* und / realisiert. Da es einfacher zu implementieren ist, benutzt dieser Taschenrechner umgekehrte polnische Notation und nicht Infix. (Umgekehrte polnische Notation wird von einigen Taschenrechnern benutzt sowie in Sprachen wie Forth und Postscript.)

In umgekehrter polnischer Notation folgt jeder Operator seinen Operanden; ein Infix-Ausdruck wie

$(1 - 2) * (4 + 5)$

wird als

$1\ 2\ -\ 4\ 5\ +\ *$

eingegeben. Klammern sind nicht notwendig; die Notation ist eindeutig, wenn wir wissen, wieviele Operanden jeder Operator erwartet.

Die Implementierung ist einfach. Jeder Operand wird auf einen Stack gebracht; kommt ein Operator an, so wird die entsprechende Anzahl Operanden (nämlich zwei bei binären Operatoren) vom Stack geholt, der Operator wird angewendet, und das Resultat wird wieder auf den Stack gebracht. Im obigen Beispiel werden 1 und 2 auf den Stack gebracht und dann durch ihre Differenz -1 ersetzt. Anschließend werden 4 und 5 auf den Stack gebracht und dann durch ihre Summe 9 ersetzt. Schließlich ersetzt das Produkt von -1 und 9, nämlich -9, diese zwei Werte auf dem Stack. Das oberste Element auf dem Stack wird entfernt und ausgegeben, wenn das Ende der Eingabezeile erreicht wird.

Als Programmstruktur ergibt sich also eine Schleife, die die richtige Operation für jeden Operator und Operanden durchführt, wenn er ankommt:

```

while ( nächster Operator oder Operand bedeutet nicht Dateiende )
    if ( Zahl )
        auf den Stack
    else if ( Operator )
        Operanden vom Stack holen
        Operation ausführen
        Resultat auf den Stack
    else if ( Zeilenende )
        Wert vom Stack holen und ausgeben
    else
        Fehler

```

Objekte auf den Stack zu bringen (*push*) und vom Stack zu holen (*pop*) ist trivial, fügt man jedoch Fehlererkennung und Fehlerbehandlung dazu, so werden diese Operationen aufwendig genug, daß man lieber jede als eigene Funktionen realisiert und nicht den Code überall im Programm dupliziert. Außerdem sollte es eine separate Funktion geben, die den nächsten Operator oder Operanden aus der Eingabe liefert.

Bisher noch nicht diskutiert wurde die wesentliche Entwurfsentscheidung: wo befindet sich der Stack, das heißt, welche Routinen greifen direkt auf ihn zu? Eine Möglichkeit ist, den Stack in **main** zu definieren, und den Stack und die momentane Position auf dem Stack an die Routinen zu übergeben, die den Stack ansprechen. Andrerseits braucht **main** nichts über die Variablen zu wissen, die den Stack kontrollieren; **main** sollte nur **push** und **pop** verwenden. Wir haben deshalb den Stack und die zugehörige Information als externe Variablen realisiert, auf die nur die Funktionen **push** und **pop**, nicht aber **main**, zugreifen können.

Diese Skizze nun in ein Programm umzusetzen, ist leicht genug. Wenn wir uns vorläufig das Programm in einer einzigen Quelldatei vorstellen, wird sie etwa so aussehen:

```

#include
#define

Funktionsdeklarationen für main
main() { ... }

externe Variablen für push und pop
void push(double f) { ... }
double pop(void) { ... }

int getop(char s[]) { ... }

von getop aufgerufene Funktionen

```

Später werden wir erklären, wie man das auf zwei oder mehr Quelldateien verteilen kann.

Die Funktion **main** besteht aus einer Schleife mit einem großen **switch**, abhängig vom Typ des Operators oder Operanden; diese Verwendung von **switch** ist wahrscheinlich typischer als das Beispiel in Abschnitt 3.4.

```

#include <stdio.h>
#include <stdlib.h> /* fuer atof() */
#define MAXOP 100 /* max. Laenge von Operand oder Operator */
#define NUMBER '0' /* Anzeige: eine Zahl wurde entdeckt */

int getop(char []);
void push(double);
double pop(void);

```

## 4.3 Externe Variablen

75

```

/* Taschenrechner mit umgekehrter polnischer Notation */
main()
{
    int type;
    double op2;
    char s[MAXOP];

    while ((type = getop(s)) != EOF) {
        switch (type) {
        case NUMBER:
            push(atof(s));
            break;
        case '+':
            push(pop() + pop());
            break;
        case '*':
            push(pop() * pop());
            break;
        case '-':
            op2 = pop();
            push(pop() - op2);
            break;
        case '/':
            op2 = pop();
            if (op2 != 0.0)
                push(pop() / op2);
            else
                printf("error: zero divisor\n");
            break;
        case '\n':
            printf("\t%.8g\n", pop());
            break;
        default:
            printf("error: unknown command %s\n", s);
            break;
        }
    }
    return 0;
}

```

Da + und \* kommutative Operatoren sind, ist die Reihenfolge irrelevant, in der die vom Stack geholten Operanden kombiniert werden, bei den Operatoren – und / müssen jedoch die rechten und linken Operanden unterschieden werden. Bei

```
push(pop() - pop()); /* FALSCH */
```

ist die Reihenfolge undefiniert, in der die zwei Aufrufe von `pop` ausgeführt werden. Um die richtige Reihenfolge zu garantieren, muß man den ersten Wert an eine temporäre Variable zuweisen, wie wir das in `main` gemacht haben.

```

#define MAXVAL 100 /* maximale Stack-Laenge */

int sp = 0;          /* naechste freie Stack-Position */
double val[MAXVAL]; /* Stack fuer die Operanden */

```

```

/* push: f auf den Stack bringen */
void push(double f)
{
    if (sp < MAXVAL)
        val[sp++] = f;
    else
        printf("error: stack full, can't push %g\n", f);
}

/* pop: Wert vom Stack holen und liefern */
double pop(void)
{
    if (sp > 0)
        return val[--sp];
    else {
        printf("error: stack empty\n");
        return 0.0;
    }
}

```

Eine Variable ist extern, wenn sie außerhalb jeder Funktion definiert wird. Also werden der Stack und der Stack-Index, die **push** und **pop** gemeinsam benutzen müssen, außerhalb von diesen Funktionen definiert. **main** selbst benutzt den Stack oder den Stack-Index nicht – die Repräsentierung kann verborgen werden.

Betrachten wir jetzt die Implementierung von **getop**, die Funktion, die den nächsten Operator oder Operanden bereitstellt. Die Aufgabe ist einfach. Zwischenräume müssen überlesen werden. Ist dann das nächste Zeichen weder eine Ziffer noch ein Dezimalpunkt, soll es als Operator geliefert werden. Andernfalls muß eine Folge von Ziffern (unter denen sich ein Dezimalpunkt befinden kann) gesammelt werden, und das Resultat ist **NUMBER**, als Hinweis, daß eine Zahl aufgefunden wurde.

```

#include <ctype.h>

int getch(void);
void ungetch(int);

/* getop: naechsten Operator oder numerischen Operanden holen */
int getop(char s[])
{
    int i, c;

    while ((s[0] = c = getch()) == ' ' || c == '\t')
        ;
    s[1] = '\0';
    if (!isdigit(c) && c != '.')
        return c; /* keine Zahl */
    i = 0;
    if (isdigit(c)) /* ganzzahligen Teil sammeln */
        while (isdigit(s[++i] = c = getch()))
            ;
    if (c == '.') /* Dezimalstellen sammeln */
        while (isdigit(s[++i] = c = getch()))
            ;
    s[i] = '\0';
    if (c != EOF)
        ungetch(c);
    return NUMBER;
}

```

## 4.3 Externe Variablen

77

Was für Funktionen sind **getch** und **ungetch**? Oft stellt ein Programm, das eine Eingabe verarbeitet, erst dann fest, daß es genügend eingelesen hat, wenn bereits zuviel eingelesen wurde. Ein Beispiel dafür ist das Sammeln einer Ziffernfolge für eine Zahl: Der Zahlenwert ist erst vollständig, wenn das erste Zeichen eingelesen wird, das keine Ziffer mehr ist. Dann hat aber das Programm bereits ein Zeichen zuviel gelesen, ein Zeichen, mit dem das Programm im Augenblick nichts anfangen kann.

Das Problem wäre einfach zu lösen, wenn man die unerwünschten Zeichen auch wieder „aus-lesen“ könnte. Immer wenn das Programm ein Zeichen zuviel liest, könnte es dieses Zeichen in die Eingabe zurückstellen, damit der Rest des Programms so tun kann, als ob dieses Zeichen noch nicht gelesen wurde. Erfreulicherweise kann man diese Operation leicht dadurch simulieren, daß man ein Paar kooperierender Funktionen schreibt. **getch** liefert das nächste Eingabezeichen, das untersucht werden soll; **ungetch** stellt Zeichen in die Eingabe zurück, die dann die nächsten Aufrufe von **getch** liefern, bevor neue Eingabe gelesen wird.

Wie diese Funktionen zusammenarbeiten, ist einfach. **ungetch** speichert die zurückgestellten Zeichen in einem gemeinsamen Puffer, einem Zeichenvektor. **getch** liest aus diesem Puffer, wenn sich dort Zeichen befinden; **getchar** wird aufgerufen, wenn der Puffer leer ist. Es muß dann noch eine Indexvariable geben, die die Position des aktuellen Zeichens im Puffer angibt.

Da der Puffer und die Indexvariable von **getch** und **ungetch** gemeinsam benutzt werden und ihre Werte zwischen Aufrufen beibehalten müssen, müssen diese Variablen extern für beide Funktionen sein. Also können wir **getch**, **ungetch** und ihre gemeinsamen Variablen folgendermaßen implementieren:

```
#define BUFSIZE 100
char buf[BUFSIZE]; /* Puffer fuer ungetch() */
int bufp = 0; /* naechste freie Position in buf */
int getch(void) /* naechstes (eventuell zurueckgestelltes) Zeichen holen */
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}
void ungetch(int c) /* Zeichen zurueckstellen */
{
    if (bufp >= BUFSIZE)
        printf("ungetch: too many characters\n");
    else
        buf[bufp++] = c;
}
```

Die Standard-Bibliothek enthält eine Funktion **ungetc**, die ein Zeichen zurückstellen kann; wir werden sie in Kapitel 7 vorstellen. Wir haben einen Vektor als Puffer benutzt und nicht nur ein einzelnes Zeichen, um einen allgemeineren Ansatz zu zeigen.

Anhang zu Aufgabe 1b – K&R Seite 80:

```
calc.h
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char []);
int getch(void);
void ungetch(int);

main.c
#include <stdio.h>
#include <stdlib.h>
#include "calc.h"
#define MAXOP 100
main() {
    ...
}

getop.c
#include <stdio.h>
#include <ctype.h>
#include "calc.h"
int getop(char s[]) {
    ...
}

stack.c
#include <stdio.h>
#include "calc.h"
#define MAXVAL 100
int sp = 0;
double val[MAXVAL];
void push(double f) {
    ...
}
double pop(void) {
    ...
}

getch.c
#include <stdio.h>
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp = 0;
int getch(void) {
    ...
}
void ungetch(int c) {
    ...
}
```

### 3. Weitere freiwillige Aufgaben

#### 3.1. Longest Line

Modularisieren Sie untenstehendes Programm.  
Teilen Sie den Code von `main.c` in die vier Dateien

1. `debug.h`  
Enthält das Debug-Makro
2. `readline.h`  
Funktionsdeklarationen für `readline(...)` und `copy(...)`.  
Idealerweise auch eine Zugriffsfunktion für die Variable `nbrOfLines`.
3. `readline.c`  
Private, globale Variable `nbrOfLines`  
Funktionsdefinitionen für `readline(...)`, `copy(...)` und `getNbrOfLines()`.
4. `newMain.c`  
Hauptprogramm mit [Debug]-Ausgaben.

`main.c`

```
/* Copyright 2016 Hochschule Luzern - Informatik */

#include <stdio.h>
#include <stdlib.h>

#ifndef DEBUG
// Macro to create debug output
#define DEBUGPRINT(_fmt, ...) \
    fprintf(stderr, "[file %s, line %d]: " \
            _fmt, __FILE__, __LINE__, __VA_ARGS__)
#else
#define DEBUGPRINT(_fmt, ...)
#endif

// Max line length
const int MAX_LINE_LENGTH = 80;

// Number of processed lines
int numberOfLines = 0;

/**
 * Read one line of input.
 */
int readLine(char s[], int limit) {
    int i = 0, c;

    for (i = 0; i < limit - 1 && (c = getchar()) != EOF && c != '\n'; i++) {
        s[i] = c;
    }

    if (c == '\n') { // Add line break character
        s[i++] = c;
    }
    s[i] = '\0'; // Add string termination character (for copy)

    numberOfLines++;

    return i; // Number of characters read
}
```

```
/**  
 * Copy a string buffer.  
 */  
char* copy(char* dest, const char* from) {  
    char* save = dest;  
  
    while (*dest++ = *from++);  
  
    return save;  
}  
  
/**  
 * Parses an input and returns the longest line.  
 */  
int main(int argc, char** argv) {  
    int len, maxLen = 0;  
    char line[MAX_LINE_LENGTH], maxLine[MAX_LINE_LENGTH];  
  
    DEBUGPRINT("Program starts\n", NULL);  
  
    while ((len = readLine(line, MAX_LINE_LENGTH)) > 0) {  
        if (len > maxLen) {  
            DEBUGPRINT("New longest line with %d characters\n", len);  
  
            maxLen = len;  
            copy(maxLine, line);  
        }  
    }  
  
    if (maxLen > 0) { // There was a longest line  
        printf("The longest line has %d characters:\n%s", maxLen, maxLine);  
    }  
    printf("Total %d lines processed.\n", numberofLines);  
  
    DEBUGPRINT("Program ends\n", NULL);  
  
    return (EXIT_SUCCESS);  
}
```

## 4. Musterlösung

### 4.1. Longest Line

```
debug.h
/* Copyright 2016 Hochschule Luzern - Informatik */

#ifndef DEBUG_H
#define DEBUG_H

/** 
 * Macro for debug aoutput.
 * The macro will be removed if DEBUG preprocessor macro is undefined.
 * @author Peter Sollberger (peter.sollberger@hslu.ch)
 */

#ifndef DEBUG
// Macro to create debug output
#define DEBUGPRINT(_fmt, ...) \
    fprintf(stderr, "[file %s, line %d]: " \
            _fmt, __FILE__, __LINE__, __VA_ARGS__)
#else
#define DEBUGPRINT(_fmt, ...)
#endif

#endif /* DEBUG_H */
```

```
readline.h
/* Copyright 2016 Hochschule Luzern - Informatik */
#ifndef READLINE_H
#define READLINE_H

/** 
 * Read a sinle line from the standard input
 * @param s      Buffer to store the line (called by reference)
 * @param limit  Size of the buffer
 * @return       Number of bytes read or EOF if read finished
 */
int readLine(char s[], int limit);

/** 
 * @return Number of lines read since start of the programm.
 */
int getNumberOfLines(void);

/** 
 * Copy a string from one buffer to another
 * @param dest Destination, must be large enough
 * @param from Source string, '\0' terminetaed
 * @return Pointer to dest
 */
char* copy(char* dest, const char* from);

#endif /* READLINE_H */
```

```
readline.c
/* Copyright 2015 Hochschule Luzern - Informatik */
#include <stdio.h>
#include "readline.h"
#include "debug.h"

static int numberOfRowsLines = 0;

int readLine(char s[], int limit) {
    int i = 0, c;

    for (i = 0; i < limit - 1 && (c = getchar()) != EOF && c != '\n'; i++) {
        s[i] = c;
    }

    if (c == '\n') { // Add new line character
        s[i++] = c;
    }
    s[i] = '\0'; // Add string end character

    numberOfRowsLines++;

    return i; // Number of characters read
}

int getNumberOfLines(void) {
    return numberOfRowsLines;
}

char* copy(char* dest, const char* from) {
    char* save = dest;

    while (*dest++ = *from++);
    return save;
}
```

```
newMain.c
/* Copyright 2016 Hochschule Luzern - Technik & Architektur */

#include <stdio.h>
#include <stdlib.h>

// Size of the line buffer
#define MAX_LINE_LENGTH 1000

#include "readline.h"
#include "debug.h"

/**
 * Parses an input and returns the longest line.
 * @author Peter Sollberger (peter.sollberger@hslu.ch)
 */
int main(int argc, char** argv) {
    int len, maxLen = 0;
    char line[MAX_LINE_LENGTH], maxLine[MAX_LINE_LENGTH];

    DEBUGPRINT("Program starts\n", NULL);

    while ((len = readLine(line, MAX_LINE_LENGTH)) > 0) {
        DEBUGPRINT("Line with length %d read\n", len);
        if (len > maxLen) {
            DEBUGPRINT("New longest line with %d characters\n", len);
            maxLen = len;
            copy(maxLine, line);
        }
    }

    if (maxLen > 0) { // There was a longest line
        printf("The longest line has %d characters:\n%s", maxLen, maxLine);
    }
    printf("Total %d lines processed.\n", getNumberOfLines());

    return (EXIT_SUCCESS);
}
```