

Programmieren mit ,C'

Manuskript zur Vorlesung Systementwicklung an der BA Stuttgart, im Fachbereich
Wirtschaftsinformatik , Herbst 2001

Autor: Alexander Kriegel, Dipl. Ing. (FH)

Inhaltsverzeichnis

1. Einleitung	4
1.1 Programmiersprachen	4
1.2 Assembler	4
1.3 Höhere Programmiersprachen	4
1.4 Die Entwicklung von C.....	4
2. Modelle	6
2.1 Flussdiagramme, Programmablaufpläne.....	6
2.2 Struktogramme	7
2.3 Blockdiagramme.....	8
2.4 Schichtenmodelle	8
2.5 Übung Taschenrechner.....	9
Modelle für einen Taschenrechner	9
3. C	10
3.1 Grundlagen.....	10
3.1.1 Hello World	10
3.1.2 Vom Sourcecode zum Programm	10
3.1.3 Compiler	10
3.1.4 Interpreter	10
3.1.5 Linker	10
3.1.6 Präprozessor, Precompiler	11
3.1.7 Übung – Hello World.....	11
3.2 Elemente eines Programms	12
3.3 Kommentare.....	13
3.4 Elementare Datentypen.....	13
3.5 Variablen	13
3.6 Konstanten	14
3.7 Operatoren	14
3.7.1 Prioritäten von Operatoren.....	15
3.7.2 True und False.....	16
3.7.3 Übung Operatoren und Prioritäten.....	16
3.8 Kontrollstrukturen	16
3.8.1 if – Bedingungen	16
3.8.2 switch - Mehrfachauswahl.....	17
3.8.3 while - Schleife.....	18
3.8.4 do while - Schleife.....	19
3.8.5 for - Schleife.....	19
3.8.6 Übung - Kontrollstrukturen und Schleifen	20
3.9 Arrays - Felder.....	20
3.9.1 Eindimensionale Arrays	20
3.9.2 Mehrdimensionale Arrays	21
3.9.3 Strings.....	21
3.9.4 Übung - Felder.....	22
3.10 Eigene Datentypen	22
3.10.1 Strukturen	22
3.10.2 Zugriff auf Strukturelemente	23
3.10.3 Unions.....	24
3.10.4 typedef	26
3.10.5 enum.....	26
3.10.6 Übung - Datentypen.....	27
3.11 Funktionen.....	27

3.11.1 return - Rückgabewerte	29
3.11.2 Parameter	30
3.11.3 Aufruf einer Funktion.....	30
3.11.4 Speicherklassen - lokale und globale Variablen	31
3.11.5 Parameter von der Kommandozeile.....	32
3.11.6 Übung - Taschenrechner mit Funktionen.....	32
3.12 Zeiger	32
3.12.1 Deklaration eines Zeigers	33
3.12.2 Der Adressoperator.....	33
3.12.3 Zugriff auf Variablen über Zeiger	33
3.12.4 Zeiger als Funktionsparameter	34
3.12.5 Speicherzugriff	35
3.12.6 Adressen von Arrays und Strings.....	35
3.12.7 Zeigerarithmetik	35
3.12.8 void Zeiger	36
3.12.9 Der Cast Operator.....	36
3.12.10 Zeiger auf Strukturen	37
3.12.11 Zeiger auf Funktionen.....	37
3.12.12 Definiton eines Funktionszeigers	38
3.12.13 Aufruf einer Funktion über Zeiger	38
3.12.14 Übung Zeiger - Mini State-Machine	39
4. Erweiterung C	39
4.1 Präprozessor	39
4.1.1 Präprozessorkonstanten.....	40
4.1.2 Präprozessoranweisungen	40
4.1.3 Headerdateien	40
4.1.4 Makros	41
4.1.5 Makros oder Funktionen	43
4.2 Arbeiten mit Dateien	43
4.2.1 FILE - Struktur.....	43
4.2.2 fopen - Öffnen einer Datei.....	44
4.2.3 fclose - Schließen von Dateien	44
4.2.4 fprintf, fscanf - Dateizugriffe	44
4.2.5 Übung - Dateien.....	45
4.3 Dynamische Speicherverwaltung	45
4.4 Kleiner Programmierknigge	46
4.5 Klassische Windowsprogrammierung.....	47
5. Ausblick.....	48
5.1 Objektorientierung	48
5.2 C++.....	49
5.3 Java.....	49
Anhänge:.....	50
A1 Zu 3.7.3. Prioritäten	52
A2 Zu 3.8.6 Kontrollstrukturen und Schleifen	52
A3 Zu 3.9.4 Felder.....	53
A4 Zu 3.10.6 Datentypen.....	54
A5 Zu 3.11.6 Taschenrechner	60
A6 Zu 3.12.14 State Machine	62
B1 ASCII Tabellen.....	67

1. Einleitung

1.1 Programmiersprachen

Computer verarbeiten Programme. Programme sind Folgen von Anweisungen in irgendeiner beliebigen Programmiersprache geschrieben. Für die Erstellung eines Programms in einer solchen Sprache gibt es eine Reihe syntaktischer Regeln, vergleichbar mit der Grammatik und der Orthographie menschlicher Sprachen. Die einzige „Sprache“ die ein Computer versteht ist die Maschinensprache. Sie besteht aus zwei Werten, 0 und 1. Man spricht von einem binären System. Dies entspricht den beiden einzigen dem Computer bekannten Zuständen, elektrische Spannung liegt an einem Bauteil an oder nicht. In einer derartigen Sprache sind keine komplexen Programme realisierbar, insbesondere eine Fehlersuche in einer fast endlosen Zeichenkette aus Nullen und Einsen ist kaum möglich. Ein binäres Zahlensystem lässt sich sehr einfach auch Hexadezimal darstellen. Dies bringt zwar etwas mehr Übersicht, doch letzten Endes handelt es sich noch immer um eine unübersichtliche Aneinanderreihung von Zahlen.

1.2 Assembler

Ob Assembler nun schon eine Programmiersprache oder noch immer Maschinencode ist, wird häufig nicht eindeutig differenziert. Assembler sind kurze Symbole wie MV A,C (Move Inhalt Register C in das Register A) oder ADD 1,2 (addiere 1 und 2). Mit derartigen Befehlen lassen sich die Vorgänge im Computer etwas veranschaulichen. Man verwendete früher zunächst Übersetzungstabellen, in welchen für jeden Assembler Befehl ein entsprechender Hexadezimalwert zugeordnet war, der in den Computer eingegeben wurde. Assembler war also eine Programmiersprache, die sozusagen nur auf dem Papier existierte. Später wurden dann Programme geschrieben, die es ermöglichten den Assemblercode einzugeben und die die Umwandlung in ‚reine‘ Maschinensprache automatisierten.

1.3 Höhere Programmiersprachen

Wenn auch Assembler einigermaßen lesbarer Code ist, ist die Erstellung großer Programme noch immer extrem aufwendig, da sich der Programmierer sehr intensiv mit computerinternen Details, wie welcher Wert steht in welcher Speicheradresse und muss in welches Register verschoben... , anstatt sich mit dem zu lösenden Problem an sich zu befassen. Dies führte zu der Entwicklung sogenannter höherer Programmiersprachen, die diese systeminternen Details vor dem Programmierer mehr oder weniger verbergen. Bekannte Vertreter dieser Sprachen sind beispielsweise Basic, Fortran, Cobol, Pascal und C. Dabei soll an dieser Stelle gleich erwähnt werden, dass C unter den höheren Sprachen noch immer eine sogenannte ‚Low Level‘ Sprache ist, da hier sehr häufig noch mit Speicheradressen etc. gearbeitet wird.

1.4 Die Entwicklung von C

C wurde ursprünglich von Denis Ritchie für das UNIX Betriebssystem DEC PDP-11 entworfen. Selbst Betriebssysteme wie Unix und Windows wurden in C geschrieben.

1978 veröffentlichten Denis Ritchie und Brian Kernighan erstmals ihr Buch "Programmieren in C". Dieses Buch galt lange Zeit als Standard und Definition der Sprache C. Inzwischen hat das American National Standards Institute (ANSI) einen offiziellen Standard geschaffen der auch internationale Anerkennung erreicht hat. Im UNIX Bereich hat auch der 1988 IEEE Std 1003.1-1988, auch bekannt als IEEE Portable Operating System Interface for Computing Environments (POSIX) große Bedeutung. ANSI und POSIX decken sich jedoch weitgehend. Diese Vorlesung behandelt den ANSI Standard. Funktionen moderner Compiler, die diesem Standard nicht entsprechen werden nicht in betracht gezogen.

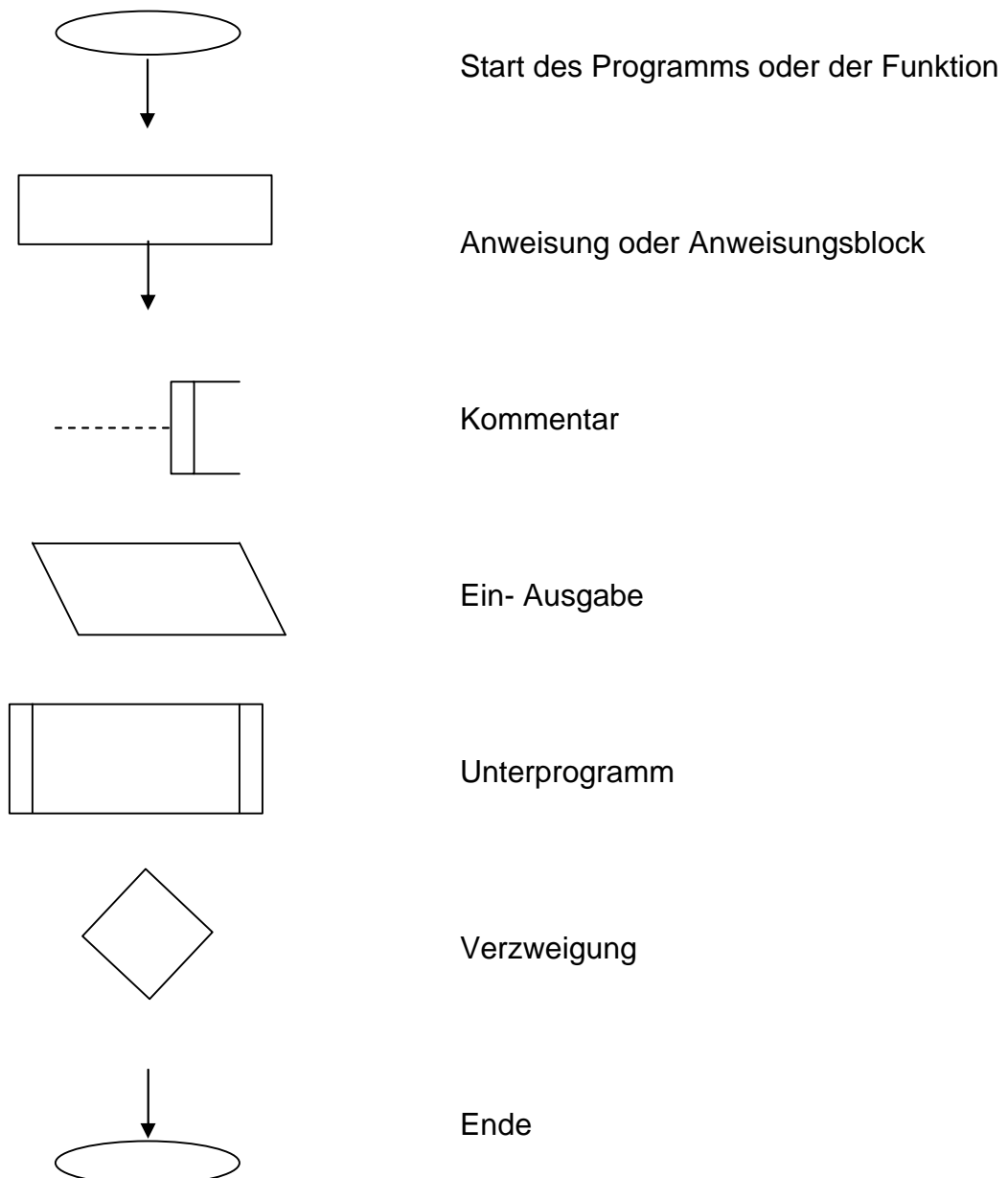
2. Modelle

Bevor mit der eigentlichen Programmierung, also dem eingeben von Code begonnen werden kann, ist es immer sinnvoll sich über den Aufbau und den Ablauf des Programms klar zu werden. Hierfür gibt es verschiedene Möglichkeiten der Modellierung.

An dieser Stelle sei darauf hingewiesen, dass sich der vermeintlich ‚zusätzliche‘ Aufwand, derartige Diagramme zu erstellen fast immer lohnt. Neben einer guten Grundlage für ein Programmdesign, erhält man auch hervorragende Dokumentationsbestandteile.

2.1 Flussdiagramme, Programmablaufpläne

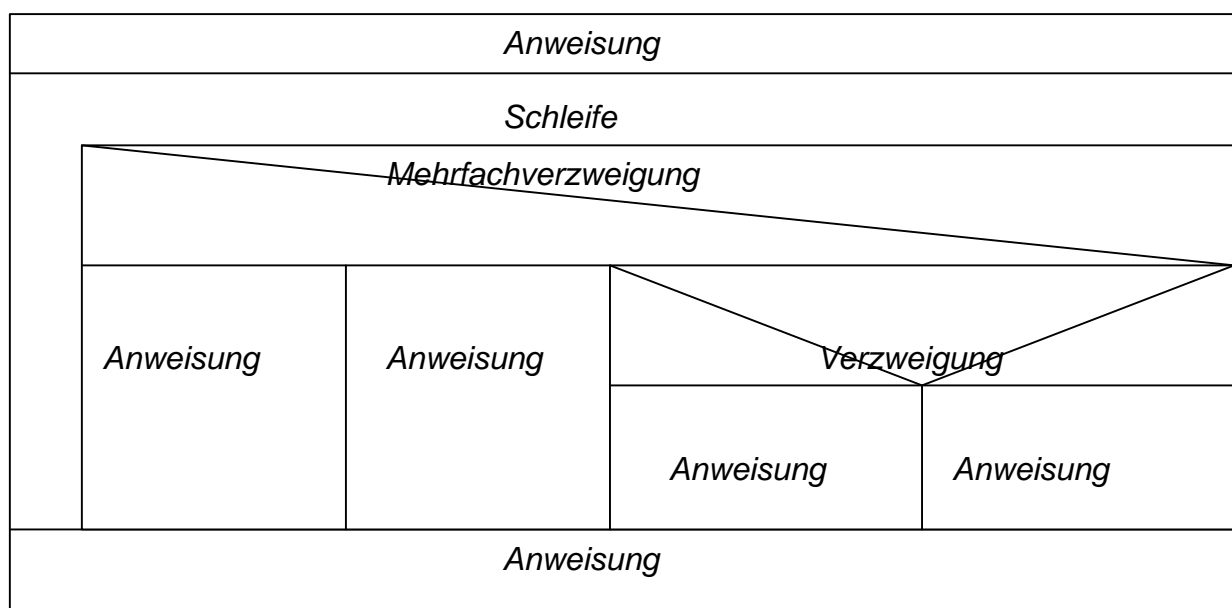
Flussdiagramme, auch Programmablaufpläne (PAP) genannt, stellen zeitliche Abläufe sehr anschaulich dar.



Man beginnt bei der Erstellung in der Regel mit grob gegliederten Flussdiagrammen, die einen Gesamtüberblick über den Ablauf der Programme geben, diese werden dann schrittweise immer weiter verfeinert. Theoretisch kann dies soweit getrieben werden, bis für jede Anweisung in der Programmiersprache ein Anweisungsblock im Diagramm vorhanden ist. Wie weit dies in der Realität dann tatsächlich Sinn macht, hängt von der Komplexität des Programms und dem Programmierer ab.

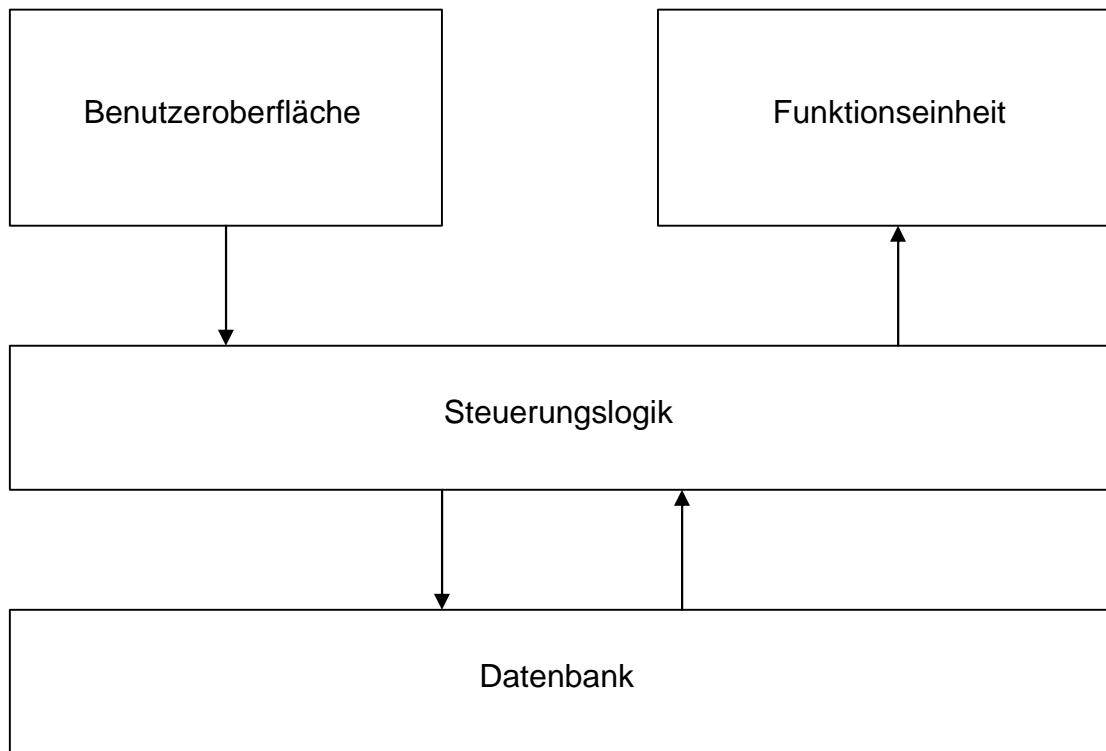
2.2 Struktogramme

Eine Alternative zu den Programmablaufplänen (PAP) bilden Struktogramme. Ihre Darstellung ist nicht ganz so anschaulich wie die der PAPs, sie sind aber durch ihre rechteckige Form geschickter in Dokumente einzubinden.



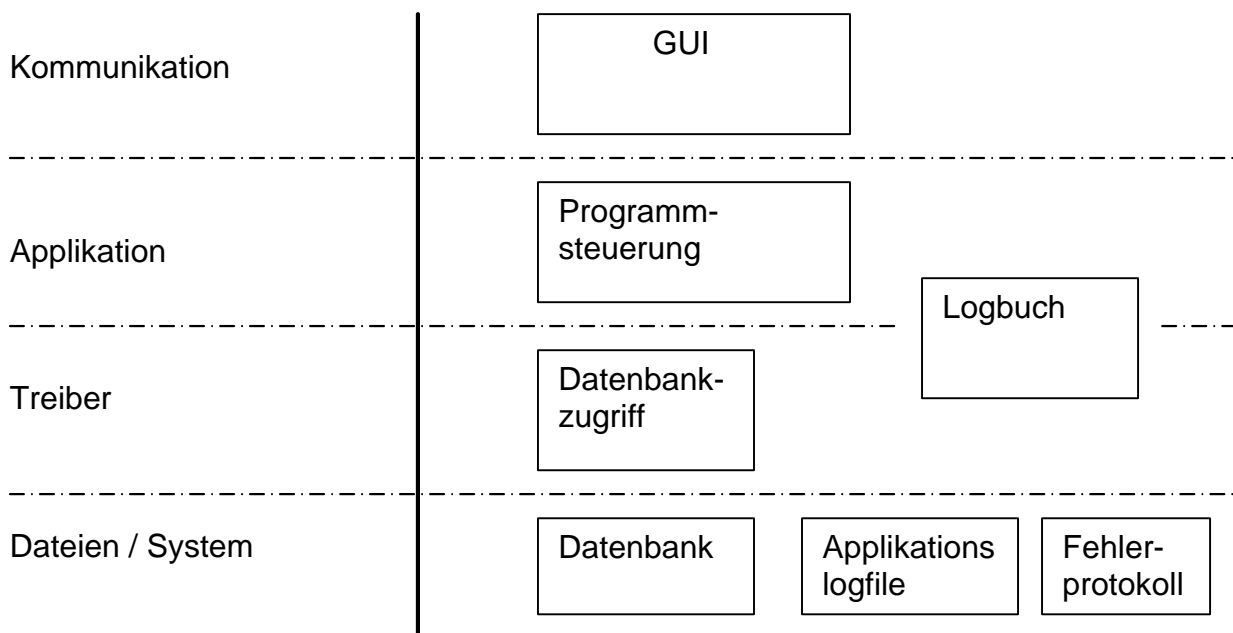
2.3 Blockdiagramme

Blockdiagramme stellen sehr komplexe Architekturen in logischen Blöcken dar.



2.4 Schichtenmodelle

Schichtenmodelle sind Modelle, die in Abstraktionsschichten gegliedert sind. An oberster Stelle steht in der Regel der Benutzer des Programms, an unterster Stelle für gewöhnlich ‚Hilfsmodule‘ wie Gerätetreiber, um die sich der Benutzer im Normalfall kaum kümmert, oder von deren Existenz er nichts weis.



2.5 Übung Taschenrechner

Modelle für einen Taschenrechner

Erstellen Sie ein Konzept für die Implementierung eines Taschenrechners, der die 4 Grundrechenarten beherrscht. Die Eingabe soll wie gewohnt, erste Zahl, Operator, zweite Zahl und abschließend ein ‚=‘ Zeichen erfolgen. Die Verarbeitung mehrerer Zahlen wird nicht gefordert.

Achten Sie schon beim Design darauf, das Programm modular aufzubauen.

3. C

3.1 Grundlagen

3.1.1 Hello World

Zunächst soll einmal das wohl berühmteste Programm der Welt, Hello World, geschrieben werden.

```
#include <stdio.h>

void main ()
{
    printf("Hello World!") ;
}
```

Den oben aufgeführten Quellcode kann man mit einem beliebigen Editor schreiben und in einer Textdatei (ASCII) mit der Extension `.c` speichern. Damit ist noch kein lauffähiges Programm entstanden, sondern nur der Quellcode für die Erstellung eines solchen.

3.1.2 Vom Sourcecode zum Programm

Damit ein Programm auf einem Rechner ausgeführt werden kann muss der Quell- oder Sourcecode in Maschinensprache übersetzt werden. Diese Arbeit übernimmt in C ein Compiler.

3.1.3 Compiler

Ein Compiler übersetzt den für den Menschen gut lesbaren Quellcode in Maschinensprache. Dabei überprüft er auch Syntaktische Fehler und meldet diese. Logische Fehler können von Compilern praktisch nicht erkannt werden. Bei der Übersetzung entsteht ein sogenannter Objektcode.

3.1.4 Interpreter

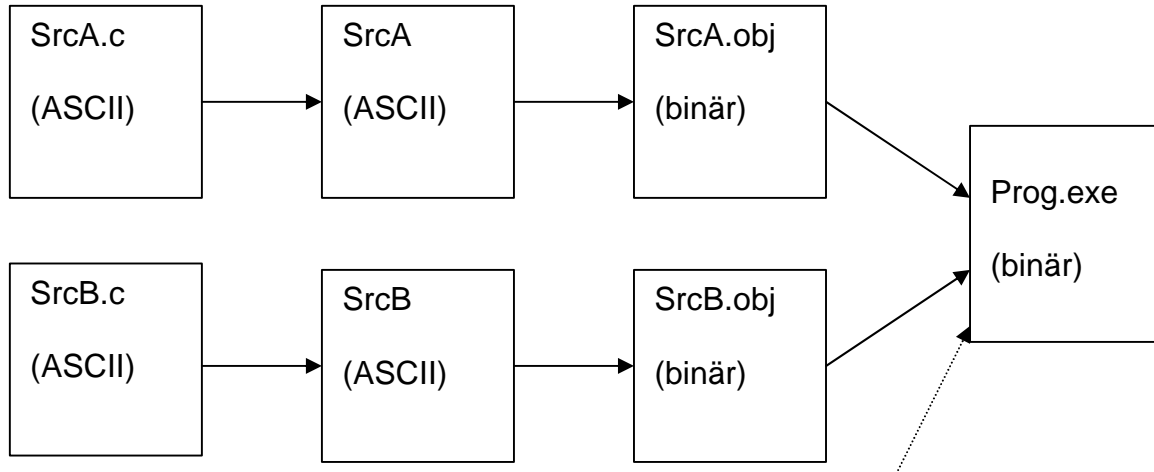
Im Unterschied zu Compilern, übersetzen Interpreter nicht das gesamte Programm, sondern lediglich das gerade auszuführende Kommando. Ein typischer Vertreter, einer interpretierten Sprache ist Basic.

3.1.5 Linker

Nachdem ein Compiler Objektcode erzeugt hat, müssen die einzelnen Module zusammengefügt werden. Diese Aufgabe übernimmt der Linker. Danach entsteht eine Datei mit der Extension `.exe` (Executable), das Programm, vollständig in Maschinensprache und daher Ausführbar.

3.1.6 Präprozessor, Precompiler

Noch bevor der C-Compiler mit der Übersetzung beginnt, wird der Quellcode vom Preprocessor bearbeitet. Dieser bearbeitet spezielle für ihn eingefügte Anweisungen wie z.B. #include, #define oder #pragma.



Bei der Verwendung moderner Entwicklungsumgebungen wie sie beispielsweise von Borland, Microsoft oder Watcom hergestellt werden, verschmelzen die einzelnen Entstehungsschritte scheinbar. Was nicht heißt, dass diese nicht mehr existent sind. Nur werden beispielsweise die nur temporär benötigten Objektdateien nach dem zusammenfügen zu einer .exe sofort wieder gelöscht. Das Wissen um diesen Vorgang ist zum Verständnis einiger Fehlermeldungen jedoch dringend nötig.

3.1.7 Übung – Hello World

Erstellen Sie ein Programm, welches den Text 'Hello World!' am Bildschirm ausgibt.

3.2 Elemente eines Programms

Nachdem nun klar ist, wie ein Programm erzeugt ist sollen die einzelnen Bestandteile eines C-Programms erläutert werden. Dazu wird noch einmal das Hello World Programm verwendet.

Headerdateien

Die erste Anweisung lautete:

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

Dies ist eine Anweisung, die den Precompiler veranlasst eine sogenannte Header-Datei einzubinden. Benötigt wird diese für die später aufgerufene printf Funktion.

Funktionen

Danach wird ein Funktion Namens main definiert. Funktionen sind Programmteile, die eine bestimmte Aufgabe übernehmen. Die Funktion main, hat eine Sonderstellung. Sie ist die Funktion mit der der Ablauf eines C-Programmes immer beginnt.

```
void main()
```

Das Schlüsselwort void besagt, dass diese Funktion keinen Rückgabewert besitzt, die leeren Klammern besagen, es werden keine Parameter übergeben. Im weiteren Verlauf wird auf Funktionen noch ausführlich eingegangen.

Eine weitere Funktion, die immer wieder im Laufe der Vorlesung benötigt wird und deshalb an dieser Stelle erwähnt werden soll ist die Funktion " printf ". Mit dieser Funktion kann ein Text, welcher als Parameter übergeben wird an der Konsole ausgegeben werden. Auf die Funktion "printf" wird im Laufe der Vorlesung noch genauer eingegangen. Zunächst genügt es, zu wissen wie "printf" zur Textausgabe verwendet werden kann:

```
printf ("Dieser Text erscheint auf der Konsole.");
```

Blöcke

In C können Anweisungen in Blöcken zusammengefasst werden. Alle Anweisungen einer Funktion, müssen beispielsweise in einem Block zusammengefasst werden. Ein Anweisungsblock beginnt mit einer geschweiften Klammer, {, und endet mit einer geschweiften Klammer, }.

Anweisungen

Das Hello World Programm enthält nur eine einzige Anweisung. Den Aufruf der Funktion printf. Diese gibt den Text 'Hello World!' am Bildschirm aus.

BEACHTEN: Alle Anweisungen werden in C mit einem Semikolon abgeschlossen!

3.3 Kommentare

Kommentare in Quelldateien sind sehr wichtig. Häufig werden diese in der Realität aus vermeintlichen Zeitgründen sträflich vernachlässigt. Darüber was ein sinnvoller und was ein überflüssiger Kommentar ist wird sicherlich nie eine einheitliche Meinung existieren. Kommentare sollen helfen, Programme zu verstehen. Dies gilt sowohl für einen anderen Programmierer, als auch für einen selber, wenn ein Programm angepasst werden muss, nachdem man den Quellcode lange nicht mehr betrachtet hat. Kommentare werden mit `/*` eingeleitet und mit `*/` beendet. Dabei können sie sich auch über mehrere Zeilen erstrecken.

```
/* Einzeiliger Kommentar */
```

```
/* Dies ist ein Kommentar,
   welcher sich über mehrere
   Zeilen erstreckt. */
```

Seit der Einführung von C++ ist auch der Doppelslash als Einzeilenkommentar verbreitet. Dieser entspricht aber nicht dem Standard ANSI C. Jedoch verstehen eigentlich alle moderne Compiler diese Form der Kommentierung.

```
// Ein C++ Kommentar, nicht Standard C!!!
```

3.4 Elementare Datentypen

In C werden Daten typisiert. Die wichtigsten Datentypen sind:

Typangabe	Breite in Bit	Wertebereich	Bemerkung
char	8	-128 bis 127	ASCII Zeichen
unsigned char	8	0 - 255	ASCII Zeichen
int	16 oder 32	-32768 bis 32767	Ganzzahl
unsigned int	16 oder 32	0 bis 65535	Vorzeichenlose Ganzzahl
long	32	-2 147 483 648 bis 2 147 483 647	Ganzzahl
unsigned long	32	0 bis 4 294 967 295	Vorzeichenlose Ganzzahl
float	32	3.4E-38 bis 3.4E23	Gleitkommazahl
double	64	1.7E-308 bis 1.7E308	Gleitkommazahl
long double	80	1.2E-4932 bis 1.2E4932	Gleitkommazahl

3.5 Variablen

Variablen bestehen aus einem Typ, einem Namen und einem Wert. Sie müssen in C immer erst Deklariert und Initialisiert werden bevor sie verwendet werden dürfen.

Deklaration einer Variablen

Bei der Deklaration werden Datentyp und Name zugewiesen:

```
int x ;
```

erzeugt eine Variable Namens x vom Typ Integer. D.h. es werden je nach System 2 oder 4 Byte Speicherplatz reserviert.

Initialisierung einer Variablen

```
x = 0 ;
```

initialisiert die x mit dem Wert 0. Häufig wird dies in einem Schritt getan:

```
int x = 0 ;
```

Verwendung von Variablen

```
int x = 5 ;
int y = 3 ;
int summe = 0 ;

summe = x + y ;
```

3.6 Konstanten

Symbolische Konstanten werden mit der Präprozessoranweisung #define definiert.

```
#define ERDUMFANG 40075.161
#define PI 3.1416
```

In allen Stellen, in denen nun die Konstante PI verwendet wird, wird der Präprozessor dieses Symbol durch 3.1416 ersetzen. Die Verwendung von symbolischen Konstanten erleichtert das Verstehen von Programmen ganz immens und verbessert die Wartbarkeit im Falle von Anpassungen.

3.7 Operatoren

In C gibt es verschiedene Arten von Operatoren:

Arithmetische Operatoren

Operator	Bedeutung
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo (Restbildung)
-	Negation

Logische Operatoren

Operator	Bedeutung
&&	Und
	Oder
!	Nicht

Bitoperatoren

Operator	Bedeutung
&	Und
	Oder
^	Exklusives Oder
~	Negation

3.7.1 Prioritäten von Operatoren

Operatoren besitzen eine Priorität die bestimmt, in welcher Reihenfolge Ausdrücke abgearbeitet werden. Im Allgemeinen gelten die uns bekannten mathematischen Regeln wie Punkt vor Strich und die der Klammersetzung.

Prioritätentabelle

Bezeichnung	Symbol	Priorität	Bewertungsreihenfolge
Klammern	() []	15	links nach rechts
Komponentenauswahl	. ->	15	links nach rechts
Unäre Operatoren			
Cast	(Datentyp)	14	rechts nach links
Größenoperator	sizeof	14	rechts nach links
Adressoperator	&	14	rechts nach links
Verweisoperator	*	14	rechts nach links
arithmetische Negation	-	14	rechts nach links
logische Negation	!	14	rechts nach links
bitlogische	~	14	rechts nach links
Inkrement	++	14	rechts nach links
Dekrement	--	14	rechts nach links
Binäre und Ternäre Operatoren			
Arithmetische	* / %	13	links nach rechts
	+ -	12	links nach rechts
Schiebeoperatoren	<< >>	11	links nach rechts
Vergleiche	> >= < <=	10	links nach rechts
	== !=	9	links nach rechts
Bitoperatoren	&	8	links nach rechts
	^	7	links nach rechts
		6	links nach rechts
Logischen	&&	5	links nach rechts
		4	links nach rechts
Bedingung (ternär)	?:	3	rechts nach links
Zuweisungen	= += -= *= /= %= >>= <<= &= ^= =	2	rechts nach links
Sequenzoperator	,	1	rechts nach links

3.7.2 True und False

Operatoren bilden zusammen mit ihren Operanten einen Ausdruck, d.h. sie repräsentieren einen Wert. Im Falle des + Operators ist dies das Ergebnis der mathematischen Operation, so steht der Ausdruck 5+3 für den gleichbedeutenden Ausdruck 8.

Im Falle der booleschen- und der Vergleichs-Operatoren wird gerne von den Ausdrücken 'true' und 'false' also 'wahr' und 'unwahr' gesprochen. Im Gegensatz zu manchen anderen (meist höheren) Programmiersprachen sind diese Ausdrücke in C nicht definiert. Es gilt aber die Vereinbarung, **jeder Wert ungleich 0 ist wahr!**

Tip: Muss in einem Programm häufig mit logischen Operationen gearbeitet werden, lohnt es sich eventuell dieses Versäumnis mit den Anweisungen

```
#define TRUE 1 und #define FALSE 0
```

nachzuholen. Eventuell ist sogar die Definition eines neuen Datentyps BOOL mit der typedef Anweisung sinnvoll.

3.7.3 Übung Operatoren und Prioritäten

Welches Ergebnis haben nachstehende Ausdrücke

a) `int x = 0, y = 2, z = 4 ;`
`x = (y+2) * z++ ;`

b) `int x = 0, y = 2, z = 4 ;`
`x = (y+2) * ++z ;`

c) `int x = 0 , y = 1; z =2 ;`
`x = z-y*x << 1 ;`

3.8 Kontrollstrukturen

Kontrollstrukturen dienen der Ablaufsteuerung eines Programms. Es werden zwei Typen unterschieden: Selektionsanweisungen (Bedingungen) und Iterationsanweisungen (Schleifen).

3.8.1 if – Bedingungen

Die if Bedingung dient dazu, eine Anweisung nur dann auszuführen, sofern eine Bedingung erfüllt ist.

Syntax:

```
if ( Bedingung ) Anweisung;
```

Es können mehrere Anweisungen zu einem Block zusammengefasst werden

```
if ( Bedingung )
{
```

```
Anweisung1 ;
Anweisung2 ;
...
Anweisung n ;
}
```

Optional kann eine else Anweisung oder ein else Block definiert werden

```
if ( Bedingung ) Anweisung ;
else Anweisung ;
```

oder

```
if ( Bedingung )
{
    Anweisung1 ;
    Anweisung2 ;
    ...
    Anweisung n ;
}
else
{
    Anweisung1 ;
    Anweisung2 ;
    ...
    Anweisung n ;
}
```

Beispiel:

```
int x = 0 ;
int y = 5 ;

if ( x > y )
{
    printf("X ist größer als Y") ;
}
else
{
    printf(„X ist nicht größer als Y“) ;
}
```

Ist x kleiner als y (vergl. Operatoren) wird der Text "X ist größer als Y" ausgegeben, in jedem anderen Falle, also wenn x größer als y ist oder beide Werte gleich sind wird der Text "X ist nicht größer als Y" .

3.8.2 switch - Mehrfachauswahl

Oft wird ein variabler Ausdruck auf seinen Wert geprüft und abhängig davon ein Programmteil ausgeführt. Dies kann natürlich mit entsprechend vielen if Bedingungen ausgewertet werden. Einfache und übersichtlicher geht dies mit der switch Bedingung.

Syntax:

```
switch ( Ausdruck )
{
    case Konstante_1 : { Anweisung(en) }
    case Konstante_2 : { Anweisung(en) }
    ...
    case Konstante_n : { Anweisung(en) }
}
```

Beispiel:

```
unsigned int x = 1;

switch ( x )
{
    case 1 : printf("x hat den Wert 1");
             break;
    case 2 : printf("x hat den Wert 2");
             break;
    case 3 : printf("x hat den Wert 3");
             break;
    default: printf ("x ist größer als 3");
}

```

Eine switch/case Anweisung sollte immer einen default Zweig beinhalten, dieser wird immer dann ausgeführt, sollte keine der anderen case Bedingungen zutreffen. Mit der break Anweisung wird ans Ende des switch Blocks gesprungen. Fehlt diese werden alle nachfolgenden Anweisungen ebenfalls durchlaufen. Dies kann beispielsweise für eine wie Konstruktion wie folgt genutzt werden.

```
char x = 'a' ;

switch ( x )
{
    case 'a':
    case 'A': printf("x ist ein großes A oder ein kleines a");
             break;
    case 'b':
    case 'B': printf("x ist ein großes B oder ein kleines b");
             break;
    default : printf("x beinhaltet weder A noch B");
}

```

3.8.3 while - Schleife

Schleifen dienen dazu einen Teil eines Programms mehrmals auszuführen. Die while Schleife prüft zu Beginn ob ein Durchlauf stattfinden soll.

Syntax:

```
while ( Bedingung )
{
    Anweisung(en) ;
}
```

Beispiel:

```
int x = 0 ;

while ( x < 10 )
{
    printf("Schleifendurchlauf");
    x++ ;
}
```

3.8.4 do while - Schleife

Im Unterschied zu der while-Schleife erfolgt die Prüfung bei der do while-Schleife erst nach dem ersten Schleifendurchlauf. D.h. die do while-Schleife wird auf jeden Fall mindestens einmal durchlaufen.

Syntax:

```
do
{
    Anweisung(en) ;
} while ( Bedingung ) ;
```

Beispiel:

```
int x = 1 ;
do
{
    x ++ ;
    printf("Schleifendurchlauf");
} while ( x < 1 ); /* Wird trotzdem einmal durchlaufen */
```

3.8.5 for - Schleife

Etwas komplexer gestaltet sich die for-Schleife, da in ihrem Schleifenkopf auch Initialisierungen getätigt werden.

Syntax:

```
for ( Initialisierung ; Bedingung ; Reinitialisierung )
{
    Anweisung(en) ;
}
```

Beispiel:

```
int x;

for ( x=0 ; x < 10 ; x++ )
{
    printf("Schleifendurchlauf");
}
```

x wird in dieser Konstruktion vor dem ersten Schleifendurchlauf auf 0 initialisiert und mit jedem Durchlauf um den Wert 1 inkrementiert, bis x nicht mehr kleiner als 10 ist.

Es können auch mehrere Initialisierungen und Reinitialisierungen erfolgen.

```
for ( i=0, y=1, z=2 ; x<y && z>0 ; x++, y-- , z-- )
```

3.8.6 Übung - Kontrollstrukturen und Schleifen

Erstellen Sie ein Programm, welches die zwei Texte, "erster Text" und "zweiter Text" 20 mal hintereinander abwechselnd ausgibt. Zur Textausgabe wird die Funktion `printf(" Ihr Text \n")` verwendet.

3.9 Arrays - Felder

In Array können Daten des gleichen Typs indiziert gespeichert werden.

3.9.1 Eindimensionale Arrays

Das einfachste Array ist eindimensional. Man kann sich ein eindimensionales Array wie eine Tabelle mit nur einer Zeile vorstellen. Die Zellen sind mit 0 beginnend durchnummeriert.

Syntax:

```
Datentyp Feldname[Elementanzahl] ;
```

Die einzelnen Feldelemente werden nun mit dem Feldnamen und dem Feldindex angesprochen, also `iFeld[0]`, `iFeld[1]` bis `iFeld[9]`. Ein `iFeld[10]` gibt es nicht da in C Indizierung prinzipiell immer mit 0 beginnen.

Beispiel:

So wird ein eindimensionales Array bestehend aus 10 Integer-Werten deklariert:

```
int iFeld[10] ;
```

Füllen eines Feldes mit den Werten 1 bis 10.

```
int iFeld[10] ;
int index ;

for ( index=0 ; index<10 ; index++ )
{
    iFeld[index] = index+1 ;
}
```

3.9.2 Mehrdimensionale Arrays

Felder können auch mehrdimensional sein. Für die Identifizierung eines Feldelements wird pro Dimension ein Index benötigt.

Syntax:

```
Datentyp Feldname [Zeilenindex] [Spaltenindex] ;
```

Beispiel

```
int iFeld [3][4] ;
```

iFeld [0] [0]	iFeld [0] [1]	iFeld [0] [2]	iFeld [0] [3]
iFeld [1] [0]	iFeld [1] [1]	iFeld [1] [2]	iFeld [1] [3]
iFeld [2] [0]	iFeld [2] [1]	iFeld [2] [2]	iFeld [2] [3]

Zweidimensionale Felder sind immer rechteckig, eine wie unten dargestellte Konstruktion ist nicht möglich:

iFeld [0] [0]	iFeld [0] [1]	iFeld [0] [2]	iFeld [0] [3]
iFeld [1] [0]	iFeld [1] [1]		
iFeld [2] [0]	iFeld [2] [1]	iFeld [2] [2]	

Felder sind in ihren Dimensionen nicht begrenzt, ein dreidimensionales Feld ist für uns noch als Würfel vorstellbar, Felder können aber n-dimensional gestaltet werden.

```
/* Achtdimensionales Feld */
int iFeld [10] [10] [10] [10] [10] [10] [10] [10] ;
```

Im Speicher werden Felder immer eindimensional (linear) abgebildet. Die einzelnen Indizes bilden in der Realität eine unterschiedliche Sprungweite. Dies wird in einem späteren Kapitel, bei der Behandlung von Zeiger von Interesse sein.

3.9.3 Strings

Strings sind Zeichenketten, also Texte bestehend aus ASCII Zeichen. Obwohl diese sehr häufig gebraucht werden, gibt es in der Sprache C leider keinen solchen Datentyp. Stings müssen daher als Character-Felder aufgebaut werden. Das letzte Zeichen ist \0, eine terminierende Null. Dies ist bei der Dimensionierung eines String-Feldes immer zu beachten. Die Deklaration

```
char text[10] ;
```

reserviert also Speicherplatz für 9 ASCII Zeichen und eine \0 (terminierende Null).

Beispiel:

Strings können mit der Funktion printf ausgegeben werden.

```
char text[6] ;

text[0] = 'H' ;
text[1] = 'a' ;
text[2] = 'l' ;
text[3] = 'l' ;
text[4] = 'o' ;
text[5] = '\0' ; /* \0 ist ein Zeichen, ASCII-Wert 0 */

printf( text ) ;
```

3.9.4 Übung - Felder

Erzeugen Sie zwei Felder, füllen Sie eines mit Ihrem Vor-, das andere mit Ihrem Nachnamen und geben Sie diese mit der Funktion printf am Bildschirm aus.

3.10 Eigene Datentypen

Bisher sind nur einfache, sogenannte primitive Datentypen wie Integer, Float oder Characters bekannt. Es ist aber in C auch möglich eigene, komplexe Datentypen zu erzeugen.

3.10.1 Strukturen

Strukturen dienen dazu, einzelne Daten zusammenzufassen. Diese sollten natürlich in einem Zusammenhang stehen, d.h. sie sollten logisch zueinander gehören. Im Unterschied zu Arrays, die nur Elemente des gleichen Datentyps aufnehmen, können Strukturen beliebig aus den primitiven Datentypen zusammengesetzt werden.

Syntax:

```
struct structname
{
    Datentyp elementname_1 ;
    Datentyp elementname_2 ;
    ...
    Datentyp elementname_n ;
} ;
```

Sehr gut lässt sich der Zweck von Strukturen an einem Beispiel erklären. C kennt keinen Datentyp der in der Lage ist eine Person, d.h. ihre relevanten Daten abzuspeichern. Soll aber ein Programm Personen verwalten können, z.B. eine Kundendatei, wäre ein derartiger Datentyp wünschenswert. Eine entsprechende Struktur könnte folgendermaßen aussehen:

Beispiel:

```
struct kunde
{
    char        vorname[20] ;
    char        nachname[20] ;
    char        geschlecht;
    unsigned int alter;
    char        strasse[20] ;
    unsigned int hausnummer ;
    unsigned int plz ;
    char        ort[20] ;
    unsigned int vorwahl ;
    unsigned int rufnummer ;
    unsigned int kundenummer ;
}
```

Strukturen können auch Strukturen als Elemente beinhalten. So wäre folgende Erweiterung denkbar:

```
struct adresse
{
    char        strasse[20] ;
    unsigned int hausnummer ;
    unsigned int plz ;
    char        ort[20] ;
    unsigned int vorwahl ;
    unsigned int rufnummer ;
}

struct kunde
{
    char        vorname[20] ;
    char        nachname[20] ;
    char        geschlecht;
    unsigned int alter;
    unsigned int kundenummer ;
    struct adresse adr ;
}
```

Letztere Konstruktion nähert sich auch modernen Programmier-Paradigmen, wie Kapselung.

Es ist zu beachten, dass bei der Instanzierung einer Strukturvariablen, das Schlüsselwort struct vorangestellt werden muss.

3.10.2 Zugriff auf Strukturelemente

Um auf Strukturelemente zuzugreifen wird der ein '.' benutzt.

Beispiel:

Gegeben sei die Struktur Kunde wie in 3.10.1 definiert.

```
struct kunde  privatKunde ;

privatKunde.alter = 18 ;
privatKunde.kundennummer = 00012001 ;
privatKunde.geschlecht = 'm' ;
```

3.10.3 Unions

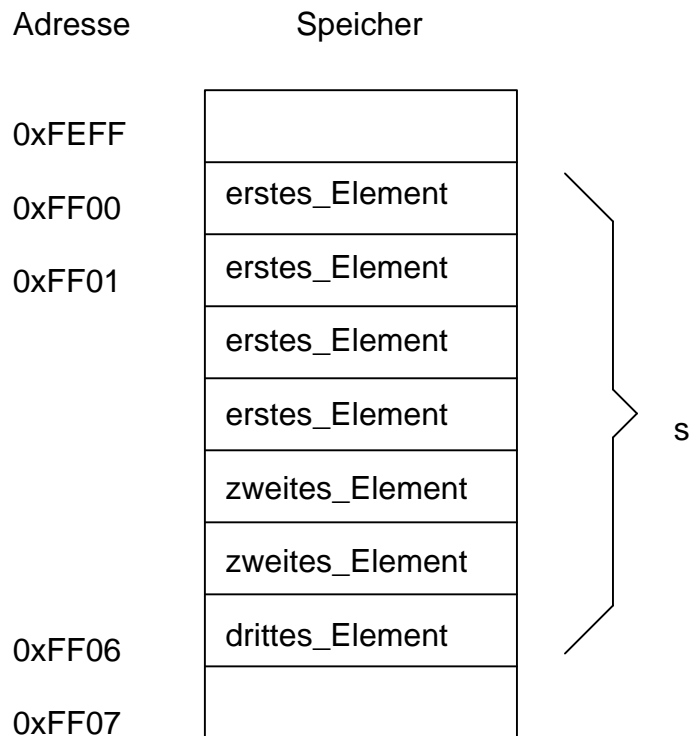
Unions sind Strukturen sehr ähnlich. Im Unterschied zu Strukturen, deren Elemente jeweils ihren eigenen Speicherplatz belegen, nutzen die Elemente einer Union den Speicherplatz gemeinsam. Dieser ist so groß wie das größte Element der Union. Die logische Konsequenz daraus ist jedoch, dass niemals auf alle Elemente gleichzeitig zugegriffen werden kann.

Ich spreche der Bedeutung von Unions in der PC- und Workstation-Welt, in der RAM-Speicher ausreichend zu Verfügung steht keine besondere Bedeutung mehr zu. Sicherlich muss in Bereichen der Mikro-Controller Programmierung auch heute noch mit Ressourcen wie Speicher besser "gehaushaltet" werden.

Beispiel:Struktur:

```
struct drei_Elemente
{
    float erstes_Element ;    /* 4 Byte */
    short zweites_Element ;  /* 2 Byte */
    char drittes_Element ;   /* 1 Byte */
} s ; /* sofortige Instanzierung */
```

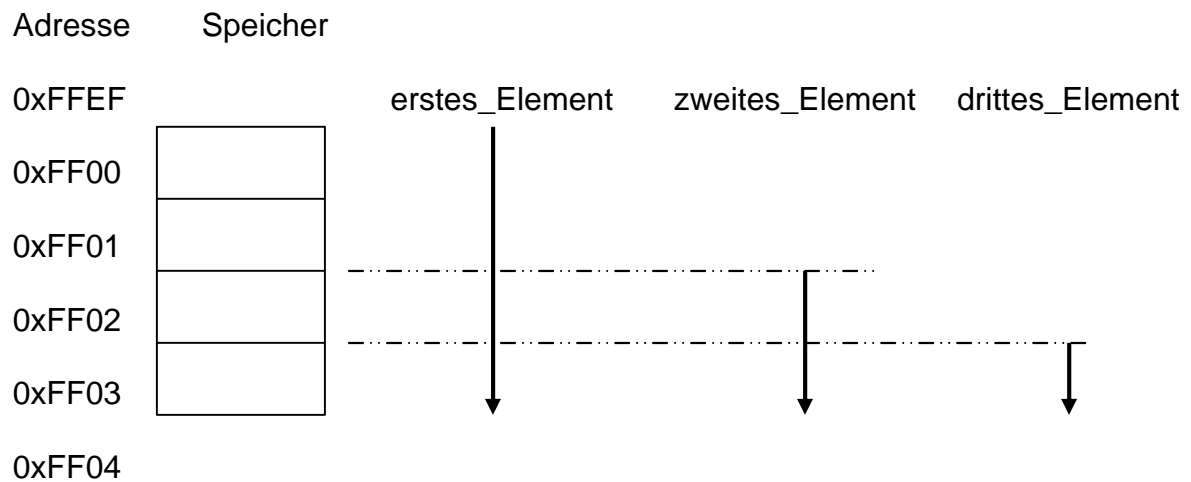
Darstellung der Struktur "s" im Speicher:



Union:

```
union dreiElemente
{
    float erstes_Element ; /* 4 Byte */
    short zweites_Element ; /* 2 Byte */
    char drittes_Element ; /* 1 Byte */
} u ; /* sofortige Instanzierung */
```

Darstellung der Union "u" im Speicher:



3.10.4 typedef

Mit typedef kann für einen Datentyp eine Ersatzbezeichnung vergeben werden. Dies ist besonders für Strukturen nützlich, da bei der Initialisierung immer zusätzlich zum Strukturnamen das Schlüsselwort struct vorangestellt werden muss.

Beispiel:

```
struct personenStruktur
{
    . . .
}
```

muss mit

```
struct prsonenStruktur person ;
```

instanziert werden.

```
typedef struct Person
{
    . . .
}
```

kann mit

```
Person person ;
```

instanziert werden.

Beispiel2:

Auch könnte auf diesem Wege ein Typ Bool definiert werden. Besser wäre hier jedoch anstelle eine Integers einen Aufzählungstyp (enum) mit den werten "true" und "false" zu verwenden.

```
typedef int Bool ;
```

3.10.5 enum

Oftmals ist nur eine eingeschränkte Anzahl von Werten für einen Datentyp gültig. Die bisher bekannten Datentypen würden den Gültigkeitsbereich u.U. nicht ausreichend einschränken. Ein Beispiel dafür wurde bereits erwähnt. Zwar konnte mit dem Schlüsselwort typedef zwar ein Datentyp Bool geschaffen werden, doch konnte sein Wertebereich nicht auf zwei Zustände eingegrenzt werden. Sind die Werte aufzählbar kann enum verwendet werden.

Beispiel:

```
enum erdteile
{
    afrika ,
    amerika,
    asien,
    australien,
    europa
};
```

mit

```
enum erdteile continent ;
```

kann eine Instanz gebildet werden. Die Variable continent kann nur die aufgeführten Werte annehmen. Eine Zuweisung wie continent = stuttgart wäre nicht möglich. Intern werden Integer-Werte vergeben. Gibt man keinen Initialwert an, wird mit der Nummerierung bei 0 begonnen, d.h. im obigen Beispiel würde die Zuweisung continent=amerika ; der Variablen continent den Wert 1 zuweisen.

3.10.6 Übung - Datentypen

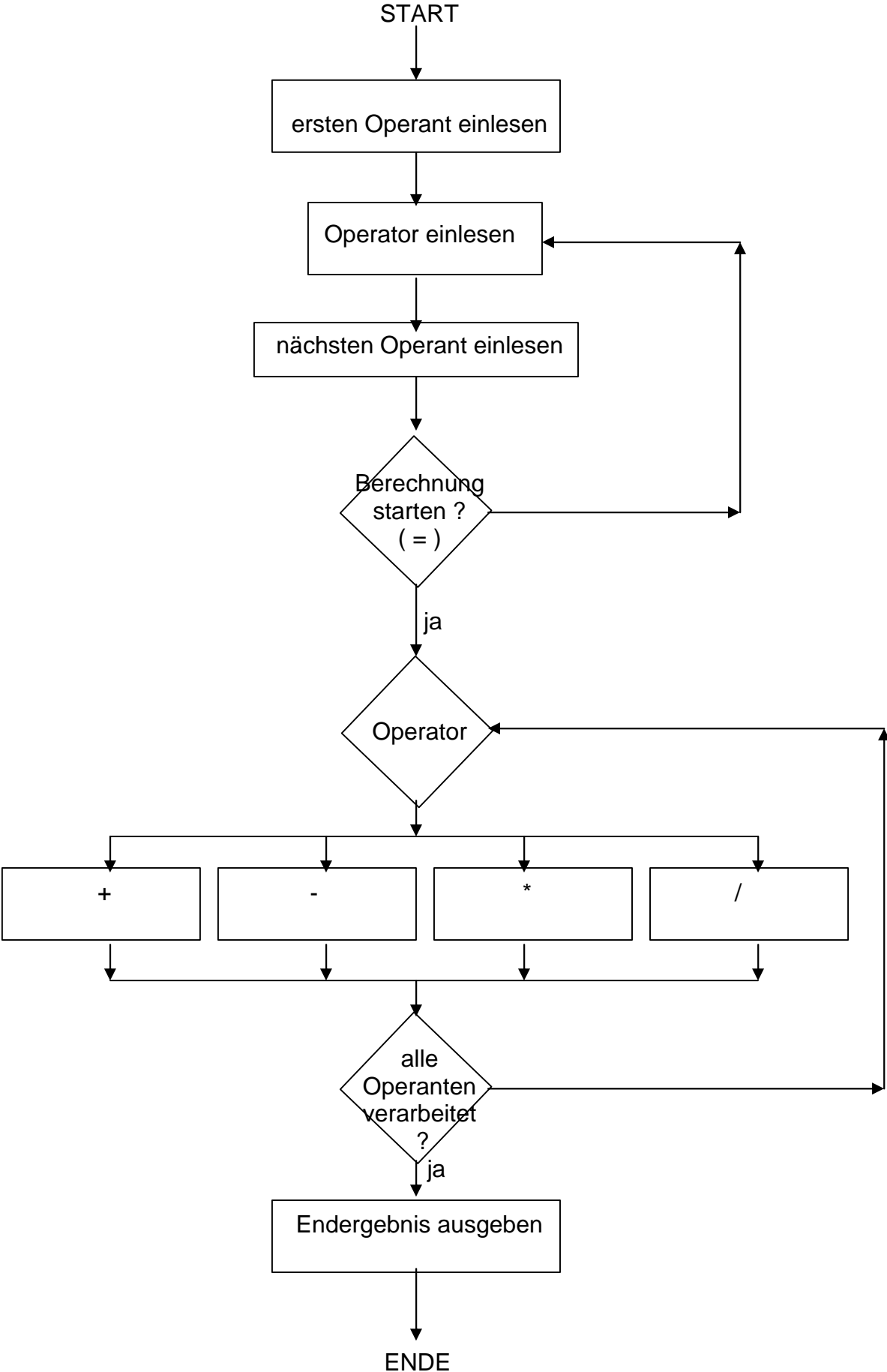
Schreiben Sie ein Programm, welches ein aus Zahlen bestehendes Datum im deutschen Formant (Tag.Monat.Jahr) in ein englisches Format (Monat, Tag Jahr) umsetzt. 14.8.2001 wird zu August, 14th 2001. Verwenden Sie Strukturen und Aufzählungstypen wo dies sinnvoll erscheint.

3.11 Funktionen

Der Sprachumfang wurde in C bewusst klein gehalten. Die eigentliche Mächtigkeit von C besteht aus den zahlreichen Bibliotheksfunktionen, die Bestandteil der C-Compiler sind. Im Kapitel 3.2 wurde beschrieben, >> *Funktionen sind Programmteile, die eine bestimmte Aufgabe übernehmen*<<. Ein Taschenrechner kann beispielsweise in folgende Einzelfunktionalitäten untergliedert werden:

1. Zahl (Operant) einlesen
2. Operator einlesen
3. Rechnung starten (=)
4. Zahlen addieren
5. Zahlen subtrahieren
6. Zahlen multiplizieren
7. Zahlen dividieren
8. Ergebnis ausgeben

Jede der obigen Funktionalitäten kann in einer Funktion durchgeführt werden. Es ist zu diesem Zeitpunkt keine Reihenfolge festgelegt oder wie oft eine dieser Funktionalitäten ausgeführt werden muss. Eine derartige Übersicht könnte in einen PAP folgendermaßen aussehen:



In diesem Beispiel wird die Funktion zum Einlesen einer Zahl zweimal benötigt, es ist aber ein und dieselbe Funktion, d.h. der Quelltext muss nur einmal geschrieben werden. Da der Taschenrechner einen komplexen Term einlesen kann, werden auch die Funktionen für die Rechenoperationen eventuell mehrfach benötigt.

Funktionen müssen wie Variablen deklariert werden, man spricht hier auch von sogenannten Prototypen.

Syntax:

```
[Speicherklasse] [Typ] Name ( [Parameterliste] ) ;
{
    [Funktionsrumpf Anweisungen] ...
}
```

Beispiel:

```
static int addiere ( int Zahl1 , int Zahl2 ) ;
{
    int erg = 0 ;
    erg = Zahl1 + Zahl2 ;
    return erg ;
}
```

Die *Speicherklasse* bestimmt ob eine Funktion nur in diesem Modul (=File) verwendet werden kann (*static*) oder auch von anderen (*extern*). Wird nichts angegeben gilt *extern*.

Der Typ einer Funktion ist gleich dem Datentyp ihres Rückgabewertes. Beachte *void*!

Jede Funktion benötigt einen eindeutigen Namen. Es gelten die gleichen Regeln wie bei der Vergabe von Variablennamen.

Über die Parameterliste werden den Funktionen Werte übergeben.

Im Funktionsrumpf oder Funktionskörper stehen die auszuführenden Anweisungen.

3.11.1 return - Rückgabewerte

Jede Funktion hat einen Rückgabewert, d.h. ein Ergebnis. Eine Funktion die beispielsweise 2 Integer-Zahlen addiert könnte das Ergebnis der Addition, ebenfalls ein Integer, als Rückgabewert besitzen.

Pascal unterscheidet beispielsweise zwischen Funktionen und Prozeduren. Prozeduren sind Unterprogramme, d.h. Programmabschnitte, die weil sie mehrfach benötigt werden zusammengefasst werden. Diese haben keinen Rückgabewert. In C gibt es nur Funktionen, diese können aber vom Typ *void* sein, d. h. sie haben dann keinen Rückgabewert.

Der Rückgabewert wird durch die return Anweisung zurückgegeben. Diese beendet auch die Ausführung einer Funktion.

3.11.2 Parameter

In den Klammern wird eine Liste mit den Parametern oder auch Argumente genannt, der Funktion übergeben. Grundsätzlich wird zwischen zwei Übergabearten unterschieden:

- a) Call by Value
- b) Call by Reference

Zunächst soll nur die erstere Version betrachtet werden, da für b) Kenntnisse über Zeiger von Nöten sind die erst in einem späteren Kapitel vermittelt werden. "Call by Value" bedeutet es wird beim Funktionsaufruf lediglich der Wert einer Variablen oder einer Konstanten an die Funktion übergeben, nicht die Variable oder Konstante selbst.

3.11.3 Aufruf einer Funktion

Eine Funktion wird über ihren Namen aufgerufen.

Beispiel:

```
#include <stdio.h>

/* Deklaration Funktionen */
int add ( int z1 , int z2 ) ;
int sub ( int z1 , int z2 ) ;
void ausgabe ( int wert ) ;

/* Definiton Funktionen */
int add ( int z1 , int z2 )
{
    int erg = z1 + z2 ;
    return erg ;
}

int sub ( int z1 , int z2 )
{
    int erg = z1 - z2 ;
    return erg ;
}

void ausgabe ( int wert )
{
    printf("Das Ergebnis ist %i", wert );
}

/*Programmstart */
void main ()
{
```

```

int a = 5 ;
int b = 3 ;
int s = 0 ;

s = add ( a , b ) ;
ausgabe ( s ) ;
s = sub ( a , b ) ;
ausgabe ( s ) ;
}

```

Beispiel2:

in diesem Beispiel soll die Bedeutung "Call bei Value" verdeutlicht werden.

```

#include <stdio.h>

void ausgabe ( int a ) ;

void ausgabe ( int a )
{
    a++ ;
    printf("a ist jetzt %i \n") ;
}

void main ( )
{
    int a = 1 ;
    printf("a ist jetzt %i \n") ;
    ausgabe ( a ) ;
    printf("a ist jetzt %i \n") ;
}

```

3.11.4 Speicherklassen - lokale und globale Variablen

Große Bedeutung kommen den Speicherklassen von Variablen zu. Es gibt Globale und Lokale Variablen. Globale Variablen sind Variablen die für ein ganzes Modul oder gar für ein ganzes Projekt gültig sind. Die Gültigkeit von lokalen Variablen beschränkt sich auf den sie umgebenen Block, in der Regel einen Funktionskörper.

Eine der wichtigsten Regeln für gute Programmierung lautet :

Programmiere immer so lokal wie möglich !

Globale Variablen werden außerhalb aller Funktionen, also auch außerhalb von main() deklariert. Eine lokale Definition bindet stärker als eine globale, d.h. wird innerhalb einer Funktion eine Variable gleichen Namens erzeugt, würde bei der Verwendung immer auf die lokale Variable referenziert werden.

Globale Variablen können innerhalb eines Moduls erzeugt werden oder aus einem anderen Modul stammen.

Beispiel:

```
/* Globale Variablen */

extern int  anzahl ;
int        anzahl2 ;

void main ( )
{
    int lokal ;
    ...
}
```

3.11.5 Parameter von der Kommandozeile

Auch die Funktion main besitzt eine Parameterliste. Diese besteht aus zwei Parametern, dem Argumentzähler und dem Argumentvektor. Der Argumentzähler, der erste Parameter muss vom Typ Integer sein. Dort wird die Anzahl der von der Kommandozeile übergebenen Parameter überreicht, die Argumente selbst stehen im Argumentvektor. Der zweite Parameter ist ein Zeiger auf ein Feld mit den Kommandozeilenparametern. Leider ist auch an dieser Stelle ein Vorgriff auf Zeiger von Nöten, daher muss die Konstruktion des Argumentvektors hier noch nicht verstanden werden.

Die Bezeichnungen argc und argv sind zwar nicht festgelegt, doch haben sich diese Namen allgemein verbreitet, so dass hier von einer Konvention gesprochen werden kann.

Beispiel:

Aufruf des Programms Test von der Kommandozeile aus:

```
C:>Test.exe Hallo Welt
```

Die main() Funktion von Test:

```
void main(int argc , char* argv[] )
{
    ...
}
```

3.11.6 Übung - Taschenrechner mit Funktionen

Realisieren Sie den Taschenrechner nach dem in Kapitel 3.11 gegebenen Flussdiagramm. Für die Eingabe der Zahlen kann die Funktion scanf verwendet werden.

3.12 Zeiger

Wenn die Sprache C so etwas wie ein Kernelement besitzt, dann sind dies Zeiger. Sie sind zweifellos das mächtigste Element dieser Sprache, doch leider ist der Umgang mit ihnen, zumindest für Einsteiger, nicht ganz einfach. Im Prinzip sind Zeiger oder Zeigervariablen ganz gewöhnliche Variablen wie ein Integer oder eine

Floatvariable. Eine Integervariable kann eine Ganzzahl, eine Floatvariable eine Gleitkommazahl aufnehmen. Eine Zeigervariable kann die Speicheradresse einer Variablen aufnehmen. Nicht mehr aber auch nicht weniger ist ein Zeiger!

3.12.1 Deklaration eines Zeigers

Zwar sind alle Zeiger gleich groß, d.h. sie belegen alle 4 Byte Speicherplatz (gilt für 32 Bit Plattformen, zu Zeiten von 16 Bit Plattformen wie DOS oder Windows 3.1 wurde noch zwischen near pointer, 2 Byte, die nur innerhalb des Speichersektors referenzieren konnten und far pointer, 4 Byte, die über den gesamten Speicherbereich reichen konnten unterschieden), doch werden sie typisiert.

Ein Zeiger wird durch ein * gekennzeichnet.

Syntax:

```
Datentyp* name ;
```

Beispiele:

```
int* pi ; /* Zeiger auf eine Integervariable */
char* pc ; /* Zeiger auf eine Charactervariable */
```

3.12.2 Der Adressoperator

Wenn ein Zeiger die Adresse einer Variablen speichern soll, muss diese zugänglich sein. Hierfür gibt es den Adressoperator &. Dieser wird einem Variablennamen vorangestellt, d.h. jetzt ist nicht wie üblich der Inhalt dieser Variablen gemeint sondern ihre Adresse.

Beispiel:

Hier werden ein Integer-, eine Float- und eine Charactervariable erzeugt, mit einem Wert gefüllt und ihre Adressen einem entsprechenden Zeiger zugewiesen.

```
int i = 100 ;
char c = 'a' ;
float f = 20.0 ;

int* pi = &i ; /*Der Zeiger pi enthält jetzt die Adresse von der Variablen i */
char* pc = &c ; /*Der Zeiger pc enthält jetzt die Adresse von der Variablen c */
float* pf = &f ; /*Der Zeiger pf enthält jetzt die Adresse von der Variablen f */
```

3.12.3 Zugriff auf Variablen über Zeiger

Es spielt keine Rolle ob über den Namen einer Variablen oder über einen Zeiger auf diese zugegriffen wird. Auf den Inhalt der Adresse auf die ein Zeiger zeigt wird ebenfalls mit dem * referenziert. Diese Doppelverwendung des * zur Deklaration von Zeigern und zur Referenzierung auf den Inhalt ist nicht sehr glücklich, aber eben nun mal in der Sprachdefinition so festgelegt.

Gegeben sei:

```
int i = 0 ;  
int* pi = &i ;
```

Die beiden Ausdrücke

```
i=5 ;      /* i wird der Wert 5 zugewiesen */
```

oder

```
*pi = 5 ; /* dem, worauf pi zeigt wird 5 zugewiesen */
```

sind gleichwertig, da pi ein Zeiger auf i ist.

3.12.4 Zeiger als Funktionsparameter

Der wohl häufigste Anwendungsfall ist die Übergabe der Adresse einer Variablen an eine Funktion, "Call by Reference", vgl. Kap. 3.11.2. Bisher wurden nur die Werte einer Variablen an eine Funktion übergeben, die Variable im Original blieb unberührt, es wurde quasi eine Kopie mit gleichen Wert erzeugt. Häufig soll aber nicht nur mit dem Wert sondern wirklich der Inhalt einer Variablen in einer Funktion geändert werden. Salopp formuliert geht dies nur auf die harte Tour, indem einfach der Inhalt der Speicherzellen, die den Wert der Variablen enthalten durch den direkten Speicherzugriff, den Zeiger ermöglichen, manipuliert wird.

Beispiel:

```
void iFunktion ( int i ) ;  
void iZeigerFunktion ( int* pi ) ;  
  
void iFunktion ( int i )  
{  
    i = 10 ;  
}  
  
void iZeigerFunktion ( int* pi )  
{  
    *pi = 15 ;  
}  
  
void main()  
{  
    int i = 0 ;  
    printf(" i hat den Wert %i \n",i ) ;  
    iFunktion( i ) ;  
    printf(" i hat den Wert %i \n",i ) ;  
    iZeigerFunktion ( &i ) ;  
    printf(" i hat den Wert %i \n",i ) ;  
}
```

3.12.5 Speicherzugriff

An dieser Stelle sei noch einmal darauf hingewiesen, dass Zeiger direkten Zugriff auf Speicheradressen ermöglichen. Selbst die Möglichkeiten des Betriebssystems diese zu überwachen sind nur eingeschränkt möglich. Die meisten der berühmten Windows Schutzverletzungen stammen durch "verbogene Zeiger" mit denen versehentlich Speicherinhalte manipuliert wurden, in welchen z.B. Daten anderer Anwendungen oder gar von Windows selbst enthalten waren. Dies geschieht in der Regel durch Fehler bei der Zeigerarithmetik.

3.12.6 Adressen von Arrays und Strings

Mit dem Adressoperator & konnte die Adresse einer Variablen ermittelt werden. Im Falle von Vektoren, Strings und Feldern ist der Name selbst die Adresse.

Beispiel:

```
/* Ein String */
char string[10] ;

char* pString ;

/* Die beiden Anweisungen sind identisch */
pString = &string[0] ;
pString = string ;
```

3.12.7 Zeigerarithmetik

Da alle Zeiger die Adresse einer Variablen speichern können, eine Adresse prinzipiell aus 4 Byte besteht, wozu benötigen Zeiger dann einen Typ? Zum einen verhindert die Typisierung das Zuweisen ungültiger Adressen. Es ist nicht möglich einem Character-Zeiger die Adresse einer Integervariablen zuzuweisen, bzw. kann dies durch die Typisierung der Zeiger vom Compiler sofort geprüft und als Fehler gemeldet werden.

Wesentlich wichtiger ist jedoch, dass mit Zeigern auch gerechnet werden kann. Es sei noch einmal darauf hingewiesen, dass Zeiger eigentlich ganz gewöhnliche Variablen sind, die lediglich als Wert eine Adresse speichern. Eine Adresse aber ist wiederum nur eine Zahl (ein Integer), meist hexadezimal angegeben. Wenn Zeiger aber ganz "normale" Variablen sind, müsste mit ihnen auch alles getan werden können, was beispielsweise auch mit einem Integer getan werden kann. Zum Beispiel eine Zahl addieren. Und dies ist zunächst auch möglich.

```
int i = 0 ;
int *pi = &i ;
```

Stünde i jetzt beispielsweise an der Adresse 0x00F0, würde *pi diesen Wert enthalten.

```
pi = pi +2 ;
```

Nach dieser Anweisung stünde in pi der Wert 0x00F2. Sicherlich macht dies auch wenn es technisch Korrekt ist so kaum Sinn. Doch eine andere Konstruktion ist denkbar:

```
int iFeld[10] = { 0,1,2,3,4,5,6,7,8,9 } ;
int* zi ;

zi = &i ; /* Beginn des Feldes */
        /*zi zeigt jetzt auf das 1. Element
        mit dem Inhalt 0*/
zi = zi + 4 /*da ein int 4 Byte groß ist, zeigt zi jetzt auf
        das zweite Element mit dem Wert 1 */
```

Durch die Typisierung wird das Rechnen mit Zeiger jedoch noch einfacher, statt

```
zi = zi + 4 ;
```

könnte zi auch einfach inkrementieren

```
zi++ ;
```

Da zi vom Typ int ist, wird der Zeiger um die Größe eines Integers, also 4 (Byte) erhöht. Wäre zi ein Zeiger auf einen double würde die gleiche Anweisung, zi++, ihn um 8 (Byte) erhöhen.

3.12.8 void Zeiger

Zeiger können auch vom Typ void Deklariert werden. Dies wird meistens dann benötigt, wenn zur Zeit der Deklaration noch nicht feststeht, auf was für einen Wert der Zeiger einmal verweisen soll. Mittels eines Cast-Operators kann der Zeiger dann später typisiert werden.

Beispiel:

```
void main()
{
    void* zeiger ;
    int i ;

    (int*) zeiger = &i ;
}
```

3.12.9 Der Cast Operator

Mit dem Cast Operator können Daten in andere Datentypen umgewandelt werden. Dies sollte jedoch immer genauestens durchdacht werden. Eine Datentypumwandlung sollte in der Regel nicht nötig sein, ausgenommen void Zeiger. Die Datentypumwandlung kann verlustbehaftet sein. Es ist recht unkritisch einen Integer in einen Float umzuwandeln, andersherum ist dies durch den Verlust der

Nachkommastellen, ein casting rundet nicht einmal mathematisch, prinzipiell Fehlerbehaftet.

Syntax:

```
Variable1 = ( Datentyp ) Variable2 ;
```

Beispiel:

```
double d;  
float f = 100.9999 ;  
  
d = (double) f ;
```

3.12.10 Zeiger auf Strukturen

Zeiger auf Strukturen können in Verbindung mit Strukturelementen benutzt werden. Bei Strukturen wird auf die einzelnen Elemente mit einem '.' referenziert. Benutzt man einen Zeiger auf eine Struktur wird dafür -> benutzt.

Beispiel:

```
#include <stdio.h>  
  
typedef struct Person  
{  
    char[10] vorname;  
    char[10] nachname;  
}  
  
void main()  
{  
    Person kunde ;  
    Person *zPerson ;  
  
    zPerson = &kunde ;  
    sprintf(zPerson->vorname, "Max" ) ;  
    sprintf(zPerson->nachname, "Muster" ) ;  
  
    printf ("%s, %s\n", zPerson->vorname , zPerson->nachname ) ;  
}
```

3.12.11 Zeiger auf Funktionen

Obwohl C-Funktionen keine variablen Datenobjekte sind, besitzen sie Variablen und Felder eine Adresse im Speicher. Wie bei Feldern und Vektoren verbirgt sich diese Adresse hinter dem Namen, dieser bezeichnet die Speicheradresse an der die Funktion mit ihren Anweisungen beginnt. Mit den Funktionsadressen lassen sich ähnliche Operationen durchführen wie mit den Adressen von Variablen. So können

diese beispielsweise einer anderen Funktion als Parameter übergeben oder in einem Zeiger gespeichert werden. Insbesondere aber lassen sich Funktionen auch über einen Zeiger, der die Startadresse enthält aufrufen.

3.12.12 Definieren eines Funktionszeigers

Syntax:

```
Datentyp (*zeigername) (Parameterliste) ;
```

Beispiel:

```
int funktion ( void ) ;

int *funktZeiger() ;

int funktion ( void )
{
    return(0);
}

void main()
{
    funktZeiger = funktion ;
}
```

3.12.13 Aufruf einer Funktion über Zeiger

Im nachfolgenden Beispiel wird die Adresse der Funktion printf aus der C Standard-Bibliothek, stdio, einem Zeiger zugewiesen und die Funktion über den Zeiger aufgerufen.

```
void main()
{
    int (*zeigerfprintf)() = printf ;
    printf("Dieser Text wurde mit printf ausgegeben.\n") ;
    (*zeigerfprintf)("Dieser text wurde über einen Zeiger auf
printf ausgegeben.\n");
}
```

Mit Zeigern auf Funktionen können sehr flexible Programme geschrieben werden. Ein gutes Beispiel ist das "Event-Handling", also das Reagieren eines Programms auf Ereignisse. Eine Funktion soll Abhängig vom Ereignis sich unterschiedlich verhalten.

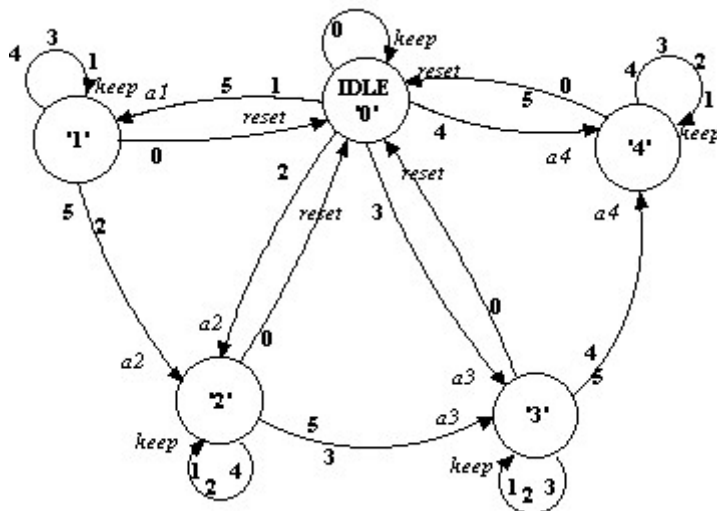
Hierzu ein Beispiel:

Ein PC soll Daten über beide seiner seriellen Schnittstellen Daten empfangen und diese in einer Datei speichern. Es gibt 4 Funktionen, Funktion 1 liest Daten von COM1 ein, Funktion 2 liest Daten vom COM2 ein. Eine dritte Funktion überwacht ob Daten auf einer Schnittstelle ankommen. Denkbar wäre, dass diese durch einen Interrupt aufgerufen wird. Die 4 Funktion schreibt die Daten, die von Funktion 1 oder

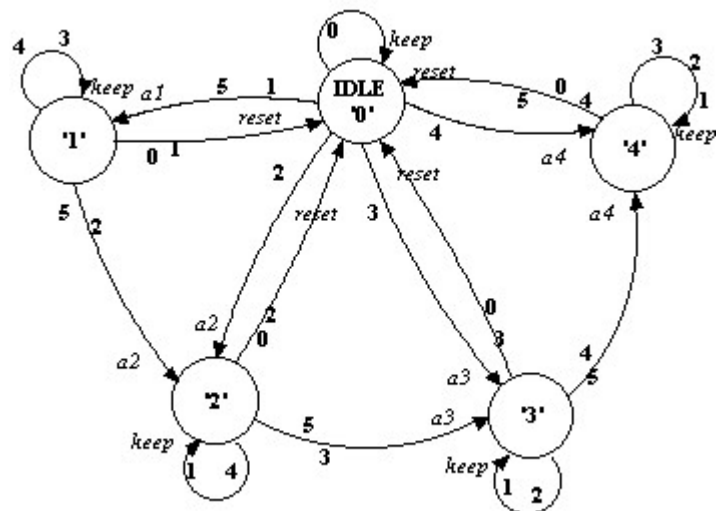
2 eingelesen werden in eine Datei. Dazu muss sie Funktion 1 oder 2 aufrufen. Als Parameter bekommt Funktion 4 einen Zeiger mit der Adresse der Funktion zum Einlesen der seriellen Schnittstelle. Sie weiß nicht von welcher Schnittstelle die Daten kommen, das ist dieser Funktion auch egal da ihre Aufgabe ausschließlich daraus besteht, die Daten einzulesen und zu speichern. Sie ruft also "Blind" die Funktion auf, auf die der Zeiger verweist.

3.12.14 Übung Zeiger - Mini State-Machine

Zur Übung soll eine State-Machine programmiert werden. Das Verhalten wird in nachfolgenden Zustandsgraphen beschrieben:



Das Programm soll anschließend geändert werden, so dass sich folgendes Verhalten der State-Machine ergibt:



4. Erweiterung C

4.1 Präprozessor

Bevor der Compiler seine Übersetzung beginnt wird der Quellcode vom Präprozessor bearbeitet. Für den Präprozessor gibt es verschiedene Anweisungen und

Konstanten. Die Anweisungen beginnen mit einem #, Konstanten beginnen und enden mit __ .

4.1.1 Präprozessorkonstanten

Konstante	Erläuterung
__LINE__	Integer: Gibt die Quelltextzeile an, die gerade bearbeitet wird.
__FILE__	String: Gibt den Namen der aktuellen Quelldatei an.
__DATE__	String: Gibt das Datum der letzten Kompilierung an.
__TIME__	String: Gibt die Uhrzeit der letzten Kompilierung an.
__STDC__	Integer: Hat den Wert 1 und existiert nur, wenn der Compiler den ANSI-C Standard einhält.

Moderne Compiler bieten in der Regel noch weitere eigene Konstanten an.

4.1.2 Präprozessoranweisungen

Anweisung	Erläuterung	
#define	Definiert eine Textersetzung für Makros und Konstanten.	
#include	Bindet eine Datei in die Aktuelle ein. Wird hauptsächlich für Headerdateien verwendet.	
#pragma	Dies sind Compilerdirektiven, also Anweisungen an den Compiler. Hinter #pragma steht die eigentliche Anweisung.	
	#pragma argsused	Verhindert die Ausgabe der Warnung"Parameter name is never used in function func-name". Häufig in Verbindung mit main() verwendet.
	#pragma comment	Kommentare in eine OBJ-Datei einfügen.
	#pragma hdrstop	Diese Direktive schließt die Liste der Header-Dateien ab, die für die Vorcompilierung vorgesehen sind. Damit kann man den für diese benötigten Plattenspeicherplatz reduzieren.
#pragma message	Damit lassen sich Nachrichten zur Kompilierung ausgeben.	
#if, #else , #endif	Bedingte Kompilierung. Damit kann die Kompilierung Abhängigkeiten unterworfen werden. So könnte ein Block A bei Verwendung eines Borland Compilers und alternativ ein etwas anders geschriebener Block B bei Verwendung eines Microsoft Compilers übersetzt werden, da diese entsprechende Präprozessorvariablen mitliefern.	
define	Ist ein Operator, in Verbindung mit #if etc, kann geprüft werden ob ein Ausdruck definiert ist oder nicht, z.B. __STDC__	
#ifdef, #ifndef	Zusammenfassung if #if define und die Negation dazu.	
#error	Damit lassen sich eigene "Compilerfehlermeldungen" ausgeben.	

4.1.3 Headerdateien

Headerdateien oder auch Includedateien genannt, sind für eine sinnvolle Programmstrukturierung sehr wichtig. In Headerdateien werden beispielsweise alle Funktionsdeklarationen, also die Prototypen der Funktionen abgelegt, die von anderen Modulen verwendet werden dürfen. Diese inkludieren dann die

entsprechende Headerdatei und die Funktion ist dort definiert und kann verwendet werden. Die wurde in diesem Kurs schon des öfteren verwendet. Durch `#include <stdio.h>` war es möglich die Funktion `printf`, deren Prototyp eben in dieser Datei definiert ist zu verwenden. Headerdateien definieren sozusagen die Schnittstelle eines Moduls. Gleiches gilt für Konstanten. Es ist sinnvoll Konstante Ausdrücke, die in mehreren Dateien benötigt werden zentral in einer Headerdatei abzulegen. Bei Änderungen müsste sonst der Ausdruck in allen Dateien gesucht und angepasst werden, was sehr zeitaufwendig und vor allem fehlerträchtig ist. In Headertateien gehören kein Funktionsimplementationen!

4.1.4 Makros

Makros stellen die Zusammenfassung von Einzelbefehlen dar. Wie symbolische Konstanten werden diese mit der `#define` Anweisung definiert. Makros können Parameter enthalten.

Makros ohne Parameter:

Syntax:

```
#define name ersatztext
```

Beispiel:

Makro zum löschen des Bildschirms:

```
#define CLS printf("\033[2J")

main()
{
    CLS ; /* ; da im Makro nicht enthalten */
    printf("Text") ;
}
```

Makros aus mehreren Anweisungen:

Makros können auch, und dann machen sie erst richtig Sinn, aus mehreren Anweisungen bestehen.

Syntax:

```
#define name { Anweisung1 ; Anweisung2 ; ... ; Anweisung n }
```

Beispiel:

```
#define ERRORHANDLER { printf("\nDivision durch 0!") ; getch() ; continue ; }
```

Um Fehler durch die Textersetzung zu verhindern, sollten Makros immer geklammert werden. Der Grund liegt bei den Prioritäten der Operatoren. Definiert man ein Makros `PREIS` und `NETTOPREIS`, könnte man in einem Programm sich folgendes vorstellen:

```
#define PREIS 11600
#define NETTOPREIS PREIS - 1600
...
summe = NETTOPREIS * stueckzahl ;
```

der Präprozessor würde folgende Textersetzung durchführen:

```
summe = 11600 - 1600 * stueckzahl ;
```

was sicher nicht das gewünschte Ergebnis liefert. Durch seine syntaktische und mathematische Korrektheit aber nicht zu einer Fehlermeldung führen würde.

```
#define NETTOPREIS (PREIS - 1600)
...
summe = (11600 - 1600) * stueckzahl ;
```

bindet das Makro entsprechend.

Makros mit Parameter:

Makros können auch Parameter übergeben bekommen.

Syntax:

```
#define name(Parameterliste ) ersatztext
```

Es ist zu beachten, dass zwischen dem Namen des Makros und den Klammern der Parameterliste ein Leerzeichen stehen darf!

Beispiel:

In nachfolgendem Beispiel wird ein Makro SUB mit zwei Parametern definiert und verwendet. Die Parameter erhalten keine Datentypen, wie es bei Funktionen der Fall wäre. Makros sind Textersetzungen, daher ist dies nicht nötig.

```
#include <stdio.h>

#define SUB( a , b ){ printf ("SUB Makro %i\n", (a-b) ); }
```

```
void main()
{
    int x,y ;
    x=10;
    y=3;

    SUB(x,y) ;
    getchar();
}
```

4.1.5 Makros oder Funktionen

Die Frage wann ein Makro und wann eine Funktion sinnvoll ist, lässt sich leider nicht beantworten. Makros haben gegenüber Funktionen einige Vorteile. Zum einem sind Makros schneller als Funktionen, da diese in den Code eingesetzt werden und nicht ein vergleichsweise aufwendiger (Bibliotheks-)Funktionsaufruf durchgeführt werden muss. Dies gilt natürlich nur, so lange die Makros selbst keine Funktionsaufrufe beinhalten. Die Routinen `getchar()` und `putchar()` sind beispielsweise als Makros implementiert um die Ein- und Ausgabe eines einzelnen Zeichens effizient zu halten. Ein weiterer Vorteil kann die Typenlosigkeit der Parameter sein. Letzterer Vorteil birgt aber auch die Gefahr von Kompatibilitätsproblemen in sich. Auch das Beispiel in 4.1.4 mit dem `NETTOPREIS`-Makro, zeigt, dass die Verwendung vom Makros durchaus ihre Tücken hat. Im Zweifelsfalle wird man in der Regel der größeren Sicherheit, also Funktionen den Vorzug geben.

4.2 Arbeiten mit Dateien

Da der Umgang mit Dateien von grundlegender Bedeutung ist, soll in diesem Kapitel etwas darauf eingegangen werden. Dateien sind in C nicht strukturiert. Ihre Daten können als eine Folge von Zeichen oder Bytes gesehen werden, also als ein Datenstrom. Jedes einzelne Zeichen, welches in einer Datei gespeichert ist, kann über seine Positionsnummer lokalisiert werden. Das erste Zeichen einer Datei hat die Positionsnummer 0, das zweite 2 usw.

C bietet in seinen Standardbibliotheken sogenannte low level und high level Funktionen zur Arbeit mit Dateien an. Unter low level Funktionen versteht man solche, die direkt auf die entsprechenden Routinen des Betriebssystems zugreifen, sogenannte system calls ausführen. C bietet aber auch eine Reihe wesentlich komfortablerer Funktionen zur Arbeit mit Dateien an.

Egal was man mit einer Datei machen möchte, es gilt folgende Vorgehensweise:

1. Datei öffnen
2. lesende oder schreibende Zugriffe auf die Datei
3. Datei schließen

Das Öffnen ermöglicht den Zugriff auf eine Datei. Schon an dieser Stelle muss festgelegt werden, wie auf die Datei zugegriffen werden soll, lesend, schreibend oder beides.

Es hängt vom Design des Programms ab, wann eine Datei wieder geschlossen wird. Solange dies nicht geschieht, kann auf die Datei immer wieder zugegriffen werden. Je nach Häufigkeit der Zugriffe ist es aus Sicherheitsgründen besser die Datei für jeden Zugriff zu öffnen und zu schließen oder sich diese Schritte zu sparen.

4.2.1 FILE - Struktur

in der `stdio`-Bibliothek ist eine Struktur namens `FILE` wie folgt definiert.

```
typedef struct
{
    char *buffer ; /*Zeiger für die Adresse des Dateipuffers */
    char *ptr;     /*Zeiger auf das nächste Zeichen im Puffer */
    int  cnt;      /*Anzahl der Zeichen im Puffer */
}
```

```

int    flags;    /*Bit mit Angaben zum Dateistatus    */
int    fd ;     /*Deskriptor (Kennzahl der Datei)    */
}
    
```

4.2.2 fopen - Öffnen einer Datei

Um eine Datei zu öffnen liefert stdio die Funktion fopen.

Prototyp:

```
FILE * fopen(char *dateiname, char *zugriffsmodus) ;
```

Im ersten Parameter wird der Dateiname übergeben. Die Datei wird sofern kein Pfad mit angegeben wird im aktuellen Verzeichnis gesucht. Bei Pfadangaben die Darstellung des \ als \\ innerhalb von Strings zu beachten!

Im zweiten Parameter wird der Zugriffsmodus übergeben.

Modus	Beschreibung
"r"	read - Datei zum Lesen öffnen. fopen liefert einen NULL.Pointer zurück, wenn die Datei nicht existiert oder nicht gefunden wurde.
"w"	write - Datei zum Schreiben öffnen. fopen liefert einen NULL.Pointer zurück, wenn die Datei nicht existiert oder nicht gefunden wurde. Existiert die Datei bereits, wird ihr Inhalt überschrieben.
"a"	append - zum Schreiben an das Dateiende öffnen. Die Datei wird erzeugt, wenn sie noch nicht existiert.
"r+"	Datei zum Lesen und Schreiben öffnen. fopen liefert einen NULL.Pointer zurück, wenn die Datei nicht existiert oder nicht gefunden wurde.
"w+"	Datei zum Lesen und Schreiben öffnen. Die Datei wird erzeugt, wenn sie noch nicht existiert. Existiert sie bereits, wird ihr Inhalt überschrieben.
"a+"	Datei zum Lesen und Anfügen öffnen. Die Datei wird erzeugt, wenn sie noch nicht existiert.

4.2.3 fclose - Schließen von Dateien

Dateien können mit fclose geschlossen werden. In der Regel wird auf das Erreichen des Zeichens EOF (End-Of-File) geprüft. fclose gibt eine 0 zurück, wenn die Datei geschlossen werden konnte, ansonsten wird ein Fehlercode zurückgegeben. Als Parameter übergibt man den Zeiger auf die FILE Struktur, welcher von fopen geliefert wurde.

Prototyp:

```
int fclose(FILE* dateizeiger) ;
```

4.2.4 fprintf, fscanf - Dateizugriffe

Es gibt mehrere Möglichkeiten in Dateien zu schreiben und aus ihnen herauszulesen. C bietet Funktionen wie fputc() und fgetc() um ein Zeichen aus einer Datei zu lesen oder zu schreiben. Alternativ gibt es dazu auch zwei Makroversionen,

getc() und putc(). Für Text bietet sich die formatierte Ein- und Ausgabe mit den Funktionen fprintf() und fscanf() an.

Prototypen:

```
int fprintf(FILE* dateizeiger, char* formatstring, ... ) ;  
int fscanf (FILE* dateizeiger, char* formatstring, ... ) ;
```

Ihr Handhabung entspricht den Funktionen printf und scanf. Intern nutzen auch printf und scanf die obigen Funktionen. Bei diesen ist der Dateizeiger dann fest auf stdin (standard in - i.d.R. die Tastatur) bzw stdout (standard out - i.d.R. die Konsole) gesetzt.

4.2.5 Übung - Dateien

Schreiben Sie zwei Programme, eines soll die Vor- und Nachnamen abfragen und diese in einer Textdatei speichern. Wird als Vor- oder Nachname "Exit" eingegeben wird das Programm beendet. Das zweite Programm liest die Datei ein und gibt diese am Bildschirm aus. Zwischen den einzelnen Namen sollen ***** zur Trennung ausgegeben werden, s.u.

Anton Aachen

Berta Berlin

Carsten Chemnitz

Dieter Dortmund

4.3 Dynamische Speicherverwaltung

Die dynamische Speicherverwaltung ist sicherlich einer der gefährlichsten und schwierigsten Bestandteile von C. Doch oftmals ist zur Zeit der Programmerstellung nicht klar, wie groß die Daten, die im Speicher abgelegt werden zur Laufzeit sind. Da RAM heute im PC Bereich nicht unbedingt knapp ist, kann man natürlich für Felder etc. einfach gigantisch großen Speicherplatz reservieren, doch ist dies a) nicht gerade elegant und egal wieviel man reserviert, irgendwann tritt dann eben doch einmal der Fall ein, dass zwei Byte mehr benötigt worden wären.

Speicher reservieren heißt Speicher allocieren. Im wesentlichen braucht man die Funktionen:

```
malloc() ;  
calloc() ;  
realloc() ;
```

und ganz wichtig:

```
free() ;
```

Prinzipiell gestaltet sich dynamische Speicherverwaltung wie folgt:

1. **Reservieren von Speicher**
`alloc()`, `malloc()` oder `realloc()`, die die gewünschte Speichergröße übergeben bekommen liefern einen Zeiger auf den Beginn des reservierten Speicherbereiches.
2. **Daten speichern**
über den von 1. gelieferten Zeiger können Daten im Speicher untergebracht werden.
3. **Speicherbereich vergrößern**
mit `realloc()` kann man sich einen neuen Speicherbereich zuweisen lassen. Ist hinter dem derzeit zugewiesenen Speicherbereich noch ausreichen Platz, so wird dieser zusätzlich reserviert, reicht dieser nicht mehr aus, wird ein entsprechend großer Bereich im Speicher gesucht und die alten Daten dorthin kopiert. Davon bemerkt der Programmierer jedoch nichts. `realloc()` liefert einen Zeiger auf den jetzt gültigen Speicherbereich.
4. **Freigeben des Speichers**
Es ist ganz wichtig den Speicher, wenn er nicht mehr benötigt wird wieder freizugeben. Ansonsten wird das System mit einem "Out of memory" Fehler zusammenbrechen.

4.4 Kleiner Programmierknigge

Die Konzeption eines Programms ist der wesentlich arbeitsintensivere Teil. Fehler im Design eines Programms sind nur sehr schwer und mit großem Aufwand wieder auszubügeln. Hier werden ein paar Regeln, Tipps, Hinweise gegeben, wie Programme zu gestalten sind.

- ? Je später Fehler entdeckt werden umso teurer, sprich zeitaufwendiger ist ihre Behebung. Syntaktische Fehler werden vom Compiler erkannt und sind daher eher unkritisch. Vermeintliche syntaktische Fehler können vom Compiler u.U. auch nicht erkannt werden, wenn wieder eine korrekte Syntax entsteht.
Die Konstruktion:

```
if ( x = 0 ) Anweisung ;
```

ist syntaktisch korrekt. `x` wird der Wert 0 zugewiesen, der Ausdruck wird geprüft, als `false` bewertet und die nachfolgende Anweisung nicht ausgeführt. Wahrscheinlich wollte der Programmierer folgendes prüfen:

```
if ( x == 0 ) Anweisung ;
```

Hätte er den Konstanten Wert vorne angestellt, würde ein Vergleich weiterhin funktionieren,

```
if ( 0 = x ) Anweisung ;
```

aber würde jeder Compiler monieren.
- ? Die Verwendung von Konstanten macht ein Programm leichter lesbarer und nachvollziehbarer. Programme werden dadurch auch leichter wartbar.
- ? Konstanten sollten zentral in einer Headerdatei definiert werden.
- ? Erst denken, dann Tippen. Eine gutes Konzept, mit Modellierungen liefert einen guten Teil Dokumentation. Ohne ausführliche Konzeption können hier schwere logische Fehler gemacht werden, die zu einem späteren Zeitpunkt kaum mehr behoben werden können.
- ? Komplexe Aufgaben Schritt für Schritt in kleine Probleme herunterbrechen, Modularisierung.

- ? Klare, gut durchdachte Schnittstellen definieren.
- ? "Keep it simple!", kleine Funktionen schreiben. Es ist schwer in 3-5 Zeilen Code einen logischen Fehler einzubauen, in großen Funktionen mit 30-50 Zeilen verliert man sehr schnell den Überblick. Eine Funktion sollte zumindest komplett auf eine Bildschirmseite passen.
- ? "Keep it simple", keine komplizierten Verschachtelungen. Dies gilt für Schleifen und Bedingungen und ganz besonders für Operatoren. Code wie dieser ist zu vermeiden:


```
x = (y*5% ++z >>3) && z ;
```
- ? "Keep it simple", lieber mal eine Klammer zuviel, wenn die einen Ausdruck übersichtlicher macht.


```
x = a*b / x+1 ;
x = ( (a*b) / x ) + 1 ;
```
- ? Dokumentation gehört in den Quellcode. Funktionen und Module direkt im Quellcode dokumentieren. Es gibt Tools, die diesen sofern ein paar Regeln eingehalten werden extrahieren können. So kann man sich zusammen mit den Plänen der Konzeption eine brauchbare technische Dokumentation erstellen lassen.

4.5 Klassische Windowsprogrammierung

Moderne Programme verlangen heute eine graphische Benutzeroberfläche. Läuft ein Programm unter Windows wird dies als selbstverständlich vorausgesetzt.

Alles was ein Anwender in Windowsprogrammen machen kann spielt sich in Fenstern ab. Er kann über Fenster Informationen an ein System schicken und Antworten erhalten. Das Verschicken und Empfangen von Nachrichten bildet die Grundlage und den großen Unterschied von Windowsanwendungen zu Konsolenprogrammen. Man kann sich vorstellen, Windowsanwendungen sind keine eigenständigen Programme, sondern nur ein Teil eines anderen größeren Programms, nämlich Windows selbst. Was auch immer im System, egal ob innerhalb von Windows oder innerhalb einer Anwendung passiert, erzeugt ein Ereignis (Event) und jedes Ereignis ergibt eine Nachricht, welche an das Betriebssystem Windows geschickt wird. Diese Nachricht enthält Informationen über das Ereignis selbst, seine Herkunft usw. So erzeugt das Bewegen der Maus eine Nachricht. Windows wertet diese Nachricht aus und reagiert beispielsweise in der Form, die Graphikkarte im PC zu veranlassen einen kleinen, weißen, schrägen Pfeil einen Pixel weiter links anzuzeigen.

Eine Windowsanwendung besteht im wesentlichen aus zwei Funktionen. Die eine, WinMain() startet und initialisiert das Programm und läuft dann in einer Endlosschleife, der Nachrichtenschleife. Diese Schleife tut nichts anderes als ständig auf das Eintreffen einer Nachricht zu warten. Geschieht dies, übergibt sie diese Nachricht an die zweite elementare Funktion von Windowsprogrammen, die Fensterfunktion. Diese besteht im Prinzip aus einer gigantische switch/case Bedingung, wobei jeder case-Zweig einen Nachrichtentyp bearbeitet. Solch eine Nachricht könnte WM_QUIT heißen, weil der Anwender einen entsprechenden Menüpunkt angeklickt hat. Im entsprechenden case-Zweig kann nun der Wunsch des Anwenders bearbeitet werden.

5. Ausblick

C hat auch heute sicherlich noch eine sehr große Bedeutung. Insbesondere Anwendungen im technischen Bereich. Dazu gehören für mich z.B. Mikrocontrollerprogramme, Maschinensteuerungen und Treibersoftware. C bildet auch die Grundlage modernerer Sprachen wie C++, C# oder Java. C ist nach Assembler die schnellste Sprache die es derzeit gibt. Der unterschied im Laufzeitverhalten zwischen C und Assembler hat einen Faktor von 1.0 bis 1.4, je nach Programmkonstellation. Im vergleich zu Java ist C etwa 10 bis 20 mal schneller.

C wird sich nicht mehr wesentlich weiterentwickeln. Die Energie der Sprachentwicklung konzentriert sich auf die moderneren Sprachen, doch da C die Grundlage für diese liefert und extrem viele Programme, die in C geschrieben wurden auf den Markt sind, ist C sicher auch in den nächsten Jahren noch ein bedeutendes Thema.

5.1 Objektorientierung

C ist von Haus aus eine prozedurale oder strukturierte Sprache. Moderne Sprachen unterstützen den Gedanken der Objektorientierung dermaßen, dass sie den Programmierer a) zwingen objektorientiert zu programmieren und b) ihm dies stark vereinfachen. Es sei aber an dieser Stelle einmal darauf hingewiesen, dass objektorientiertes programmieren (OOP) nur ein Denkansatz ist. Dieser ist im übrigen viel älter als objektorientierte Sprachen und durchaus auch in C und anderen Sprachen umsetzbar. Sehr gute Programmierer haben dies schon lange vor C++ und SmallTalk getan. Nur ist dies leider in C sehr unangenehm zu Handhaben.

Was ist OOP?

Es ist an dieser Stelle auch nicht annähernd möglich eine vollständige Einführung in objektorientierte Programmierung zu geben. Lediglich ein paar Worte um eine Vorstellung dessen zu vermitteln, was sich hinter dem Begriff verbirgt.

Die strukturierte Programmierung hat einige Nachteile. Viele Funktionen werden mehrfach in unterschiedlichen Programmen benötigt. Werden Funktionen entsprechend abstrahiert, d.h. in kleine Basisbestandteile aufgeteilt, können diese vielleicht zum Teil wiederverwendet werden. Meist gelingt dies aber nicht. Oft passen die Funktionen nicht 100%ig und es ist ärgerlich, dass sie neu geschrieben werden müssen, obwohl sie doch fast dasselbe tun.

Die größte Schwierigkeit jedoch bereitet das Abbilden realer Probleme oder Anforderungen in ein prozedurales System. Dies ist deshalb so schwierig, weil unsere Welt nicht prozedural sondern objektorientiert ist.

Was sind Objekte?

Was sind Objekte in der realen Welt? Dinge, also alles was wir irgendwie erfassen können. Objekte besitzen Eigenschaften (Attribute) und Methoden (Fähigkeiten). Ein Auto ist ein Objekt. Es hat eine Reihe von Eigenschaften die es identifizierbar machen. Zum Beispiel eine Farbe, ein Leergewicht etc. Es hat aber auch Fähigkeiten. Es kann beschleunigen, lenken oder bremsen. Objekte werden in Klassen beschrieben.

Was sind Klassen?

Klassen beschreiben Objekte. Sie sind als solche nicht existent. Jedes Objekt braucht eine Klasse die es beschreibt. Die Klasse wird instanziiert, d.h. eine Instanz(=Objekt) wird gebildet.

Die Grundpfeiler der objektorientierten Programmierung

- ? Orientierung an der Natur (Realität)
- ? Vererbung
- ? Kapselung
- ? Information Hiding
- ? Polymorphismus

5.2 C++

Mit C++ wurde aus dem prozeduralen C eine objektorientierte Sprache. C++ beinhaltet alles was C kann. Programme können auch hybrid geschrieben werden, d.h. es könnte beispielsweise eine grafische Oberfläche komfortabel in C++ implementiert werden, während ein zugehöriges Treibermodul für Datenbank- oder Hardwarezugriffe aus Effizienzgründen in Standard C geschrieben werden könnte.

C++ entspricht in seiner Syntax noch immer C. D.h. eine for-Schleife in C++ sieht genauso aus wie eine in C usw. Neu sind natürlich einige Befehle wie "class" die der OO zugehörig sind. Bei sehr tiefgehender Betrachtung kann aber beispielsweise eine Klasse auf eine C-Struktur reduziert werden.

5.3 Java

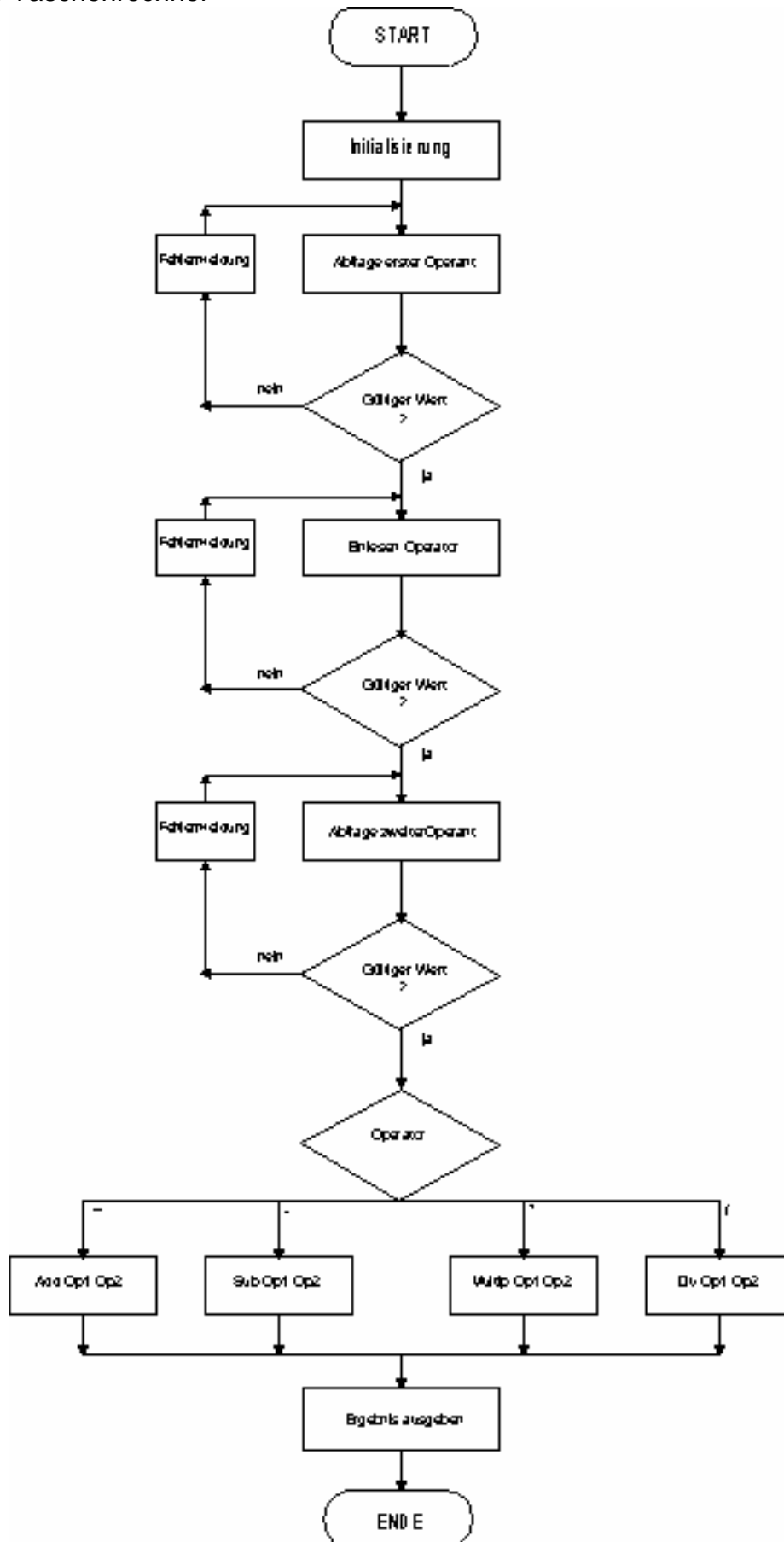
Java gewann seine Bedeutung mit dem Internet, da es die Softwareentwicklung für diese Umgebung erst richtig ermöglicht. Keine Sprache bietet derzeit eine umfangreichere Basis an, auf der für das WWW, eMail usw. entwickelt werden kann. Ob C# hier Java den Boden abgräbt bleibt zum heutigen Zeitpunkt noch abzuwarten. Java hat viele Aspekte die in C/C++ schwierig zu handhaben sind vereinfacht. So ist die dynamische Speicherverwaltung und Threadsynchronisation fast automatisiert worden. Nicht alle Vereinfachungen konnten kompromisslos vollzogen werden. Zeigerarithmetik gibt es in Java nicht mehr. Zeiger selbst existieren zwar weiterhin, doch werden diese weitgehend vor dem Programmierer verborgen.

Java ist die erste wirklich plattformunabhängige Programmiersprache. Reines ANSI C wird zwar auch von allen bedeutenden Plattformen unterstützt, doch diese Unabhängigkeit existiert nur auf Quellcodeebene. Objectfiles und Executables müssen für jede Plattform erzeugt werden. Kompilierte Javaklassen sind auf jeder Plattform die eine entsprechende Runtimeumgebung (Virtual Machine) anbieten lauffähig.

Anhänge:

- A - Lösungen der Aufgaben aus dem Skript
- B - ASCII Tabelle

A1 Zu 2.5 Taschenrechner



A1 Zu 3.7.3. Prioritäten

- a) 16
- b) 20
- c) 4 oder 5 (es ist nicht normiert, womit bei der Verwendung von Schiebeoperatoren aufgefüllt wird. Borland C++ mit ANSI benutzt 0)

A2 Zu 3.8.6 Kontrollstrukturen und Schleifen

```
void main(void)
{
    int index=0;

    while ( index < 20 )
    {
        index++ ;
        printf("%i.\t",index);
        printf("Erster Text.\n");
        printf("%i.\t",index);
        printf("Zweiter Text.\n");
        getchar() ;
    }
}
```

oder besser, da flexibel:

```
void main(void)
{
    int index=0;

    while ( index < 20 )
    {
        index++ ;
        if ( index%2 ) /* Teiler bestimmt an welcher Stelle
                       "Zweiter Text" ausgegeben wird */
        {
            printf("%i.\t",index);
            printf("Erster Text.\n");
            getchar() ;
        }
        else
        {
            printf("%i.\t",index);
            printf("Zweiter Text.\n");
            getchar() ;
        }
    }
}
```

A3 Zu 3.9.4 Felder

Lösung 1:

```
#include <stdio.h>
#pragma argsused /*Borland Compiler */

void main()
{
    char vorname[] = {
'A','l','e','x','a','n','d','e','r','\0' } ;
    char nachname[] = { 'K','r','i','e','g','e','l','\0' } ;

    printf("%s %s", vorname, nachname);
    getchar() ; /* nur wegen DOS-Box */
}
```

Lösung 2:

```
void main()
{
    char vorname[32] ;
    char nachname[32];

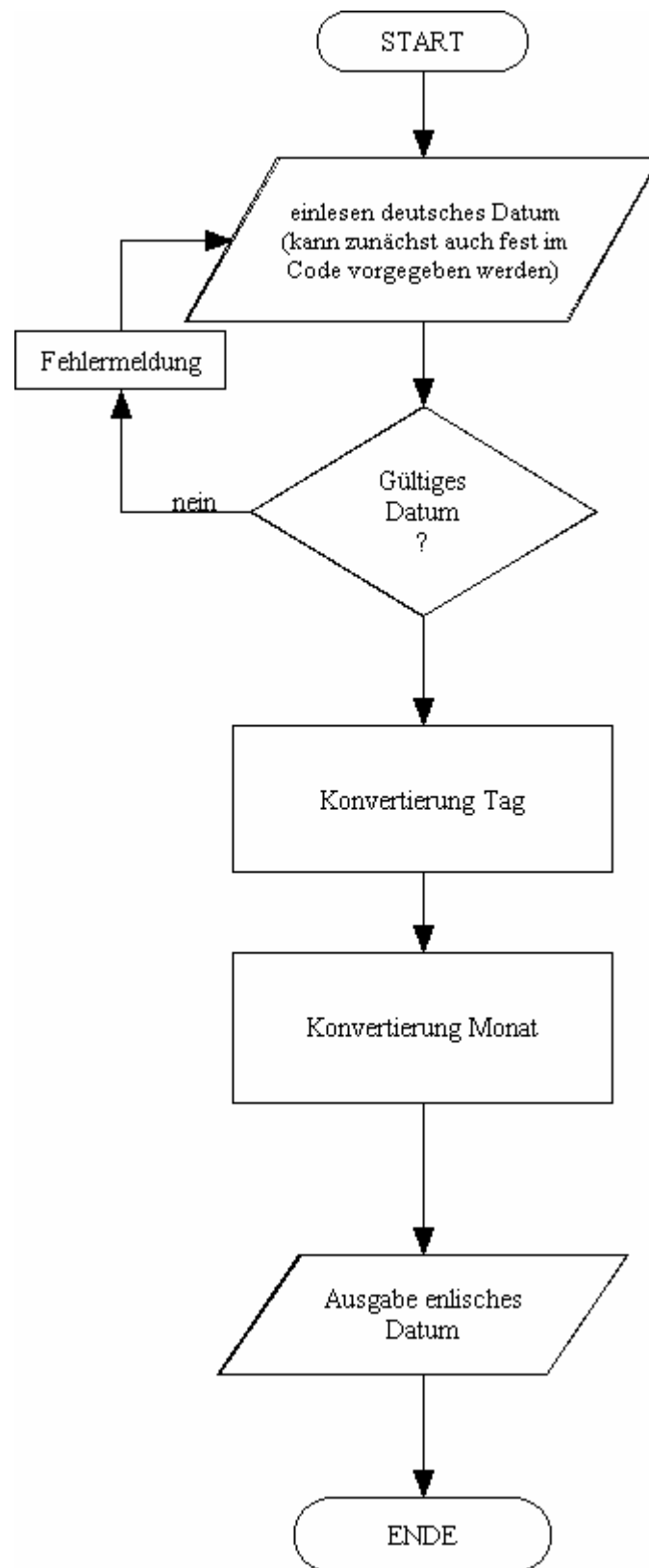
    vorname[0] = 'A' ;
    vorname[1] = 'l' ;
    vorname[2] = 'e' ;
    vorname[3] = 'x' ;
    vorname[4] = 'a' ;
    vorname[5] = 'n' ;
    vorname[6] = 'd' ;
    vorname[7] = 'e' ;
    vorname[8] = 'r' ;
    vorname[9] = '\0' ;

    nachname[0] = 'K' ;
    nachname[1] = 'r' ;
    nachname[2] = 'i' ;
    nachname[3] = 'e' ;
    nachname[4] = 'g' ;
    nachname[5] = 'e' ;
    nachname[6] = 'l' ;
    nachname[7] = '\0' ;

    printf("%s %s", vorname, nachname);
    getchar() ;
}
```

A4 Zu 3.10.6 Datentypen

Flussdiagramm:



Quellcode:

```

#include <stdio.h>
#pragma argsused

void main()
{
    /* Datendefinitionen */

    struct dtDat
    {
        int tag ;
        int monat ;
        int jahr ;
    } dtDat;

    struct enDat
    {
        char day[5] ;
        char month[10] ;
        int year ;
    } enDat ;

    /* Dateninitialisierung */

    dtDat.tag = 0 ;
    dtDat.monat = 0 ;
    dtDat.jahr = 0 ;

    enDat.day[0] = '0' ;
    enDat.day[1] = '\0' ;
    enDat.month[0] = '0' ;
    enDat.month[1] = '\0' ;
    enDat.year = 0 ;

    /* deutsches Datum */
    printf("Tag? ");
    scanf("%i",&dtDat.tag) ; /* sollte überprüft werden (z.B. if ... ) */
    printf("Monat? ");
    scanf("%i",&dtDat.monat) ; /* sollte überprüft werden (z.B. if ... ) */
    printf("Jahr? ");
    scanf("%i",&dtDat.jahr) ; /* sollte überprüft werden (z.B. if ... ) */
    /*
    dtDat.tag = 14 ;
    dtDat.monat = 8 ;
    dtDat.jahr = 2001 ;
    */

    /* Konvertierung Tag */
    switch ( dtDat.tag )
    {
        case 1 : enDat.day[0] = '1' ;
                enDat.day[1] = 's' ;
                enDat.day[2] = 't' ;
                enDat.day[3] = '\0' ;
                break ;

        case 2 : enDat.day[0] = '2' ;
                enDat.day[1] = 'n' ;
                enDat.day[2] = 'd' ;
                enDat.day[3] = '\0' ;
                break ;

        case 3 : enDat.day[0] = '3' ;
                enDat.day[1] = 'r' ;
                enDat.day[2] = 'd' ;
                enDat.day[3] = '\0' ;
                break ;

        case 4 : enDat.day[0] = '4' ;
                enDat.day[1] = 't' ;
                enDat.day[2] = 'h' ;
                enDat.day[3] = '\0' ;
                break ;

        case 5 : enDat.day[0] = '5' ;
                enDat.day[1] = 't' ;
                enDat.day[2] = 'h' ;
                enDat.day[3] = '\0' ;
                break ;

        case 6 : enDat.day[0] = '6' ;
    }
}

```

```

        enDat.day[1] = 't' ;
        enDat.day[2] = 'h' ;
        enDat.day[3] = '\0' ;
        break ;
    case 7 : enDat.day[0] = '7' ;
            enDat.day[1] = 't' ;
            enDat.day[2] = 'h' ;
            enDat.day[3] = '\0' ;
            break ;
    case 8 : enDat.day[0] = '8' ;
            enDat.day[1] = 't' ;
            enDat.day[2] = 'h' ;
            enDat.day[3] = '\0' ;
            break ;
    case 9 : enDat.day[0] = '9' ;
            enDat.day[1] = 't' ;
            enDat.day[2] = 'h' ;
            enDat.day[3] = '\0' ;
            break ;
    case 10: enDat.day[0] = '1' ;
            enDat.day[1] = '0' ;
            enDat.day[2] = 't' ;
            enDat.day[3] = 'h' ;
            enDat.day[4] = '\0' ;
            break ;
    case 11: enDat.day[0] = '1' ;
            enDat.day[1] = '1' ;
            enDat.day[2] = 'n' ;
            enDat.day[3] = 'd' ;
            enDat.day[4] = '\0' ;
            break ;
    case 12: enDat.day[0] = '1' ;
            enDat.day[1] = '2' ;
            enDat.day[2] = 'v' ;
            enDat.day[3] = 'e' ;
            enDat.day[4] = '\0' ;
            break ;
    case 13: enDat.day[0] = '1' ;
            enDat.day[1] = '3' ;
            enDat.day[2] = 't' ;
            enDat.day[3] = 'h' ;
            enDat.day[4] = '\0' ;
            break ;
    case 14: enDat.day[0] = '1' ;
            enDat.day[1] = '4' ;
            enDat.day[2] = 't' ;
            enDat.day[3] = 'h' ;
            enDat.day[4] = '\0' ;
            break ;
    case 15: enDat.day[0] = '1' ;
            enDat.day[1] = '5' ;
            enDat.day[2] = 't' ;
            enDat.day[3] = 'h' ;
            enDat.day[4] = '\0' ;
            break ;
    case 16: enDat.day[0] = '1' ;
            enDat.day[1] = '6' ;
            enDat.day[2] = 't' ;
            enDat.day[3] = 'h' ;
            enDat.day[4] = '\0' ;
            break ;
    case 17: enDat.day[0] = '1' ;
            enDat.day[1] = '7' ;
            enDat.day[2] = 't' ;
            enDat.day[3] = 'h' ;
            enDat.day[4] = '\0' ;
            break ;
    case 18: enDat.day[0] = '1' ;
            enDat.day[1] = '8' ;
            enDat.day[2] = 't' ;
            enDat.day[3] = 'h' ;
            enDat.day[4] = '\0' ;
            break ;
    case 19: enDat.day[0] = '1' ;
            enDat.day[1] = '9' ;
            enDat.day[2] = 't' ;
            enDat.day[3] = 'h' ;
            enDat.day[4] = '\0' ;

```

```

        break ;
case 20: enDat.day[0] = '2' ;
        enDat.day[1] = '0' ;
        enDat.day[2] = 't' ;
        enDat.day[3] = 'h' ;
        enDat.day[4] = '\0' ;
        break ;
case 21: enDat.day[0] = '2' ;
        enDat.day[1] = '1' ;
        enDat.day[2] = 's' ;
        enDat.day[3] = 't' ;
        enDat.day[4] = '\0' ;
        break ;
case 22: enDat.day[0] = '2' ;
        enDat.day[1] = '2' ;
        enDat.day[2] = 'n' ;
        enDat.day[3] = 'd' ;
        enDat.day[4] = '\0' ;
        break ;
case 23: enDat.day[0] = '2' ;
        enDat.day[1] = '3' ;
        enDat.day[2] = 'r' ;
        enDat.day[3] = 'd' ;
        enDat.day[4] = '\0' ;
        break ;
case 24: enDat.day[0] = '2' ;
        enDat.day[1] = '4' ;
        enDat.day[2] = 't' ;
        enDat.day[3] = 'h' ;
        enDat.day[4] = '\0' ;
        break ;
case 25: enDat.day[0] = '2' ;
        enDat.day[1] = '5' ;
        enDat.day[2] = 't' ;
        enDat.day[3] = 'h' ;
        enDat.day[4] = '\0' ;
        break ;
case 26: enDat.day[0] = '2' ;
        enDat.day[1] = '6' ;
        enDat.day[2] = 't' ;
        enDat.day[3] = 'h' ;
        enDat.day[4] = '\0' ;
        break ;
case 27: enDat.day[0] = '2' ;
        enDat.day[1] = '7' ;
        enDat.day[2] = 't' ;
        enDat.day[3] = 'h' ;
        enDat.day[4] = '\0' ;
        break ;
case 28: enDat.day[0] = '2' ;
        enDat.day[1] = '8' ;
        enDat.day[2] = 't' ;
        enDat.day[3] = 'h' ;
        enDat.day[4] = '\0' ;
        break ;
case 29: enDat.day[0] = '2' ;
        enDat.day[1] = '9' ;
        enDat.day[2] = 't' ;
        enDat.day[3] = 'h' ;
        enDat.day[4] = '\0' ;
        break ;
case 30: enDat.day[0] = '3' ;
        enDat.day[1] = '0' ;
        enDat.day[2] = 't' ;
        enDat.day[3] = 'h' ;
        enDat.day[4] = '\0' ;
        break ;
case 31: enDat.day[0] = '3' ;
        enDat.day[1] = '1' ;
        enDat.day[2] = 's' ;
        enDat.day[3] = 't' ;
        enDat.day[4] = '\0' ;
        break ;
default: enDat.day[0] = 'e' ;
        enDat.day[1] = 'r' ;
        enDat.day[2] = 'r' ;
        enDat.day[3] = '\0' ;
        break ;

```

```

}

/* Konvertierung Monat */
switch ( dtDat.monat )
{
    case 1 : enDat.month[0] = 'J' ;
             enDat.month[1] = 'a' ;
             enDat.month[2] = 'n' ;
             enDat.month[3] = 'u' ;
             enDat.month[4] = 'a' ;
             enDat.month[5] = 'r' ;
             enDat.month[6] = 'y' ;
             enDat.month[7] = '\0' ;
             break ;
    case 2 : enDat.month[0] = 'F' ;
             enDat.month[1] = 'e' ;
             enDat.month[2] = 'b' ;
             enDat.month[3] = 'r' ;
             enDat.month[4] = 'u' ;
             enDat.month[5] = 'a' ;
             enDat.month[6] = 'r' ;
             enDat.month[7] = 'y' ;
             enDat.month[8] = '\0' ;
             break ;
    case 3 : enDat.month[0] = 'M' ;
             enDat.month[1] = 'a' ;
             enDat.month[2] = 'r' ;
             enDat.month[3] = 'c' ;
             enDat.month[4] = 'h' ;
             enDat.month[5] = '\0' ;
             break ;
    case 4 : enDat.month[0] = 'A' ;
             enDat.month[1] = 'p' ;
             enDat.month[2] = 'r' ;
             enDat.month[3] = 'i' ;
             enDat.month[4] = 'l' ;
             enDat.month[5] = '\0' ;
             break ;
    case 5 : enDat.month[0] = 'M' ;
             enDat.month[1] = 'a' ;
             enDat.month[2] = 'y' ;
             enDat.month[3] = '\0' ;
             break ;
    case 6 : enDat.month[0] = 'J' ;
             enDat.month[1] = 'u' ;
             enDat.month[2] = 'n' ;
             enDat.month[3] = 'e' ;
             enDat.month[4] = '\0' ;
             break ;
    case 7 : enDat.month[0] = 'J' ;
             enDat.month[1] = 'u' ;
             enDat.month[2] = 'l' ;
             enDat.month[3] = 'y' ;
             enDat.month[4] = '\0' ;
             break ;
    case 8 : enDat.month[0] = 'A' ;
             enDat.month[1] = 'u' ;
             enDat.month[2] = 'g' ;
             enDat.month[3] = 'u' ;
             enDat.month[4] = 's' ;
             enDat.month[5] = 't' ;
             enDat.month[6] = '\0' ;
             break ;
    case 9 : enDat.month[0] = 'S' ;
             enDat.month[1] = 'e' ;
             enDat.month[2] = 'p' ;
             enDat.month[3] = 't' ;
             enDat.month[4] = 'e' ;
             enDat.month[5] = 'm' ;
             enDat.month[6] = 'b' ;
             enDat.month[7] = 'e' ;
             enDat.month[8] = 'r' ;
             enDat.month[9] = '\0' ;
             break ;
    case 10: enDat.month[0] = 'O' ;
             enDat.month[1] = 'c' ;
             enDat.month[2] = 't' ;
             enDat.month[3] = 'o' ;

```

```

        enDat.month[4] = 'b' ;
        enDat.month[5] = 'e' ;
        enDat.month[6] = 'r' ;
        enDat.month[7] = '\\0' ;
        break ;
    case 11: enDat.month[0] = 'N' ;
            enDat.month[1] = 'o' ;
            enDat.month[2] = 'v' ;
            enDat.month[3] = 'e' ;
            enDat.month[4] = 'm' ;
            enDat.month[5] = 'b' ;
            enDat.month[6] = 'e' ;
            enDat.month[7] = 'r' ;
            enDat.month[8] = '\\0' ;
            break ;
    case 12: enDat.month[0] = 'D' ;
            enDat.month[1] = 'e' ;
            enDat.month[2] = 'c' ;
            enDat.month[3] = 'e' ;
            enDat.month[4] = 'm' ;
            enDat.month[5] = 'b' ;
            enDat.month[6] = 'e' ;
            enDat.month[7] = 'r' ;
            enDat.month[8] = '\\0' ;
            break ;
    default: enDat.month[0] = 'e' ;
            enDat.month[1] = 'r' ;
            enDat.month[2] = 'r' ;
            enDat.month[3] = '\\0' ;
            break ;
}

/* Konvertierung Jahr ( 1:1 ) */
enDat.year = dtDat.jahr ;

/* Ausgabe englisches Datum */

printf("\\n%s, %s %i",enDat.month, enDat.day, enDat.year);
getchar(); /* Return moech im Tasaturpuffer */
getchar();
}

```

A5 Zu 3.11.6 Taschenrechner

```

#pragma hdrstop
#pragma argsused

#include <stdio.h>

/* Globals */
char charInput ;
char zeichen[5] ;
float floatInput , fMem ;
float zahlen[5] ;
int index = 0 ;

/* Function Prototypes */
int vGetInput() ;
float add(float s1, float s2);
float sub(float s1, float s2);
float mul(float s1, float s2);
float div(float s1, float s2);

/* Main Function */
void main()
{
    do
    {
        } while ( vGetInput() ) ;

    index = 0 ;
    floatInput = zahlen[0] ;

    while ( zeichen[index] != '=' )
    {
        switch ( zeichen[index] )
        {
            case '+':    fMem = zahlen[index+1] ;
                        floatInput = add(floatInput, fMem) ;
                        break ;
            case '-':    fMem = zahlen[index+1] ;
                        floatInput = sub(floatInput, fMem) ;
                        break ;
            case '*':    fMem = zahlen[index+1] ;
                        floatInput = mul(floatInput, fMem) ;
                        break ;
            case '/':    fMem = zahlen[index+1] ;
                        floatInput = div(floatInput, fMem) ;
                        break ;
            default:    printf("Error");
                        break ;
        }
        index ++ ;
    }

    printf("Ergebnis = %f \n",floatInput);
    getchar();
}

/*****
/* Implementations
*****/
int vGetInput()
{
    printf("Zahl: ");
    scanf("%f",&floatInput);
    _flushall();
    zahlen[index] = floatInput;
    printf("Operator: ");
    scanf("%c",&charInput);
    _flushall();
    zeichen[index] = charInput;
    index ++;
    if (index > 5)
    {
        printf("Feldueberlauf!\n");
    }
}

```

```

        zeichen[index-1] = '=' ;
        return 0 ;
    }
    if (zeichen[index-1] == '=' )
    {
        return 0 ;
    }
    return 1 ;
}

/*****/
float add(float s1, float s2)
{
    return s1 + s2 ;
}

/*****/
float sub(float s1, float s2)
{
    return s1 - s2 ;
}

/*****/
float mul(float s1, float s2)
{
    return s1 * s2 ;
}

/*****/
float div(float s1, float s2)
{
    return s1 / s2 ;
}

```

A6 Zu 3.12.14 State Machine

Lösung Teil 1 :

```

#pragma hdrstop
#pragma argsused

#include <stdio.h>
#include <stddef.h> /* NULL */

/* definitions and constants */
#define ESC_KEY 27
/* States */
#define STATE_COUNT 5
#define IDLE 0
#define STATE1 1
#define STATE2 2
#define STATE3 3
#define STATE4 4

/*Events*/
#define EVENT_COUNT 6
#define RESET 0
#define EVENT1 1
#define EVENT2 2
#define EVENT3 3
#define EVENT4 4
#define NEXT 5

/*Datatypes*/
typedef struct
{
    int state ;
    int nextState ;
    void (*action)() ;
} StaMaElement ;

/*Globals*/
StaMaElement currentSme ;

/* function prototypes */
/*State Machine actions*/
void action1() ;
void action2() ;
void action3() ;
void action4() ;
void keep();
void reset();
/*others*/
int getAction() ;

/* main */
void main(void)
{
    int in ;
    StaMaElement sme[STATE_COUNT][EVENT_COUNT] ;

    /*Initialize */
    /*Idle */
    sme[IDLE][RESET].state = IDLE ;
    sme[IDLE][EVENT1].state = IDLE ;
    sme[IDLE][EVENT2].state = IDLE ;
    sme[IDLE][EVENT3].state = IDLE ;
    sme[IDLE][EVENT4].state = IDLE ;
    sme[IDLE][NEXT].state = IDLE ;
    sme[IDLE][RESET].action = keep ;
    sme[IDLE][EVENT1].action = action1 ;
    sme[IDLE][EVENT2].action = action2 ;
    sme[IDLE][EVENT3].action = action3 ;
    sme[IDLE][EVENT4].action = action4 ;
    sme[IDLE][NEXT].action = action1 ;
    sme[IDLE][RESET].nextState = IDLE ;
    sme[IDLE][EVENT1].nextState = STATE1 ;
    sme[IDLE][EVENT2].nextState = STATE2 ;
    sme[IDLE][EVENT3].nextState = STATE3 ;

```

```

sme[IDLE][EVENT4].nextState = STATE4 ;
sme[IDLE][NEXT].nextState  = STATE1 ;
/*State1*/
sme[STATE1][RESET].state   = STATE1 ;
sme[STATE1][EVENT1].state  = STATE1 ;
sme[STATE1][EVENT2].state  = STATE1 ;
sme[STATE1][EVENT3].state  = STATE1 ;
sme[STATE1][EVENT4].state  = STATE1 ;
sme[STATE1][NEXT].state    = STATE1 ;
sme[STATE1][RESET].action  = reset ;
sme[STATE1][EVENT1].action = keep ;
sme[STATE1][EVENT2].action = action2 ;
sme[STATE1][EVENT3].action = keep ;
sme[STATE1][EVENT4].action = keep ;
sme[STATE1][NEXT].action   = action2 ;
sme[STATE1][RESET].nextState = IDLE ;
sme[STATE1][EVENT1].nextState = STATE1 ;
sme[STATE1][EVENT2].nextState = STATE2 ;
sme[STATE1][EVENT3].nextState = STATE1 ;
sme[STATE1][EVENT4].nextState = STATE1 ;
sme[STATE1][NEXT].nextState  = STATE2 ;
/*State2*/
sme[STATE2][RESET].state   = STATE2 ;
sme[STATE2][EVENT1].state  = STATE2 ;
sme[STATE2][EVENT2].state  = STATE2 ;
sme[STATE2][EVENT3].state  = STATE2 ;
sme[STATE2][EVENT4].state  = STATE2 ;
sme[STATE2][NEXT].state    = STATE2 ;
sme[STATE2][RESET].action  = reset ;
sme[STATE2][EVENT1].action = keep ;
sme[STATE2][EVENT2].action = keep ;
sme[STATE2][EVENT3].action = action3 ;
sme[STATE2][EVENT4].action = keep ;
sme[STATE2][NEXT].action   = action3 ;
sme[STATE2][RESET].nextState = IDLE ;
sme[STATE2][EVENT1].nextState = STATE2 ;
sme[STATE2][EVENT2].nextState = STATE2 ;
sme[STATE2][EVENT3].nextState = STATE3 ;
sme[STATE2][EVENT4].nextState = STATE2 ;
sme[STATE2][NEXT].nextState  = STATE3 ;
/*State3*/
sme[STATE3][RESET].state   = STATE3 ;
sme[STATE3][EVENT1].state  = STATE3 ;
sme[STATE3][EVENT2].state  = STATE3 ;
sme[STATE3][EVENT3].state  = STATE3 ;
sme[STATE3][EVENT4].state  = STATE3 ;
sme[STATE3][NEXT].state    = STATE3 ;
sme[STATE3][RESET].action  = reset ;
sme[STATE3][EVENT1].action = keep ;
sme[STATE3][EVENT2].action = keep ;
sme[STATE3][EVENT3].action = keep ;
sme[STATE3][EVENT4].action = action4 ;
sme[STATE3][NEXT].action   = action4 ;
sme[STATE3][RESET].nextState = IDLE ;
sme[STATE3][EVENT1].nextState = STATE3 ;
sme[STATE3][EVENT2].nextState = STATE3 ;
sme[STATE3][EVENT3].nextState = STATE3 ;
sme[STATE3][EVENT4].nextState = STATE4 ;
sme[STATE3][NEXT].nextState  = STATE4 ;
/*State4*/
sme[STATE4][RESET].state   = STATE4 ;
sme[STATE4][EVENT1].state  = STATE4 ;
sme[STATE4][EVENT2].state  = STATE4 ;
sme[STATE4][EVENT3].state  = STATE4 ;
sme[STATE4][EVENT4].state  = STATE4 ;
sme[STATE4][NEXT].state    = STATE4 ;
sme[STATE4][RESET].action  = reset ;
sme[STATE4][EVENT1].action = keep ;
sme[STATE4][EVENT2].action = keep ;
sme[STATE4][EVENT3].action = keep ;
sme[STATE4][EVENT4].action = keep ;
sme[STATE4][NEXT].action   = reset ;
sme[STATE4][RESET].nextState = IDLE ;
sme[STATE4][EVENT1].nextState = STATE4 ;
sme[STATE4][EVENT2].nextState = STATE4 ;
sme[STATE4][EVENT3].nextState = STATE4 ;
sme[STATE4][EVENT4].nextState = STATE4 ;
sme[STATE4][NEXT].nextState  = IDLE ;

```

```

currentSme = sme[0][0] ;

printf("\n\n\n") ;
printf ("State machine initialized\n*****\n\n");
printf ("Valid actions are 1,2,3 and 4\n0 will reset state machine\n\n");
printf ("Press ESC to exit programm.\n\n");

/* run */
do
{
    in = getAction() ;

    switch ( in )
    {
        case RESET: currentSme = sme[currentSme.state][RESET] ;
                    break ;
        case EVENT1: currentSme = sme[currentSme.state][EVENT1] ;
                    break ;
        case EVENT2: currentSme = sme[currentSme.state][EVENT2] ;
                    break ;
        case EVENT3: currentSme = sme[currentSme.state][EVENT3] ;
                    break ;
        case EVENT4: currentSme = sme[currentSme.state][EVENT4] ;
                    break ;
        case NEXT:  currentSme = sme[currentSme.state][NEXT] ;
                    break ;
        default:    break ;
    }
    /*print last State */
    printf("State is: %i\n",currentSme.state);
    /*set next state */
    currentSme.state=currentSme.nextState ;
    /* do action */
    currentSme.action() ;
} while ( in != ESC_KEY );
}

/* functions */
void action1()
{
    printf("Executing ACTION 1 ... \n");
    printf("Current State is: %i\n\n",currentSme.state);
}
void action2()
{
    printf("Executing ACTION 2 ... \n");
    printf("Current State is: %i\n\n",currentSme.state);
}
void action3()
{
    printf("Executing ACTION 3 ... \n");
    printf("Current State is: %i\n\n",currentSme.state);
}
void action4()
{
    printf("Executing ACTION 4 ... \n");
    printf("Current State is: %i\n\n",currentSme.state);
}
void keep()
{
    printf("No action to execute, KEEP ... \n");
    printf("Current State is: %i\n\n",currentSme.state);
}
void reset()
{
    printf("Reseting State Machine.\n");
    printf("Current State is: %i\n\n",currentSme.state);
}

int getAction()
{
    int buffer ;
    buffer = getchar();
    flushall() ;
    if (buffer == ESC_KEY ) return ESC_KEY ;
    /* convert ASCII Code to integer */
    buffer = buffer - 48 ;
}

```

```

    if (buffer < 0 || buffer > 6 ) return 0 ;
    return buffer ;
}

```

Lösung Teil 2:

```

...
/*Initialize */
/*Idle */
sme[IDLE][RESET].state = IDLE ;
sme[IDLE][EVENT1].state = IDLE ;
sme[IDLE][EVENT2].state = IDLE ;
sme[IDLE][EVENT3].state = IDLE ;
sme[IDLE][EVENT4].state = IDLE ;
sme[IDLE][NEXT].state = IDLE ;
sme[IDLE][RESET].action = keep ;
sme[IDLE][EVENT1].action = action1 ;
sme[IDLE][EVENT2].action = action2 ;
sme[IDLE][EVENT3].action = action3 ;
sme[IDLE][EVENT4].action = action4 ;
sme[IDLE][NEXT].action = action1 ;
sme[IDLE][RESET].nextState = IDLE ;
sme[IDLE][EVENT1].nextState = STATE1 ;
sme[IDLE][EVENT2].nextState = STATE2 ;
sme[IDLE][EVENT3].nextState = STATE3 ;
sme[IDLE][EVENT4].nextState = STATE4 ;
sme[IDLE][NEXT].nextState = STATE1 ;
/*State1*/
sme[STATE1][RESET].state = STATE1 ;
sme[STATE1][EVENT1].state = STATE1 ;
sme[STATE1][EVENT2].state = STATE1 ;
sme[STATE1][EVENT3].state = STATE1 ;
sme[STATE1][EVENT4].state = STATE1 ;
sme[STATE1][NEXT].state = STATE1 ;
sme[STATE1][RESET].action = reset ;
sme[STATE1][EVENT1].action = reset ; /**/
sme[STATE1][EVENT2].action = action2 ;
sme[STATE1][EVENT3].action = keep ;
sme[STATE1][EVENT4].action = keep ;
sme[STATE1][NEXT].action = action2 ;
sme[STATE1][RESET].nextState = IDLE ;
sme[STATE1][EVENT1].nextState = IDLE ; /**/
sme[STATE1][EVENT2].nextState = STATE1 ;
sme[STATE1][EVENT3].nextState = STATE1 ;
sme[STATE1][EVENT4].nextState = STATE1 ;
sme[STATE1][NEXT].nextState = STATE2 ;
/*State2*/
sme[STATE2][RESET].state = STATE2 ;
sme[STATE2][EVENT1].state = STATE2 ;
sme[STATE2][EVENT2].state = STATE2 ;
sme[STATE2][EVENT3].state = STATE2 ;
sme[STATE2][EVENT4].state = STATE2 ;
sme[STATE2][NEXT].state = STATE2 ;
sme[STATE2][RESET].action = reset ;
sme[STATE2][EVENT1].action = keep ;
sme[STATE2][EVENT2].action = reset ; /**/
sme[STATE2][EVENT3].action = action3 ;
sme[STATE2][EVENT4].action = keep ;
sme[STATE2][NEXT].action = action3 ;
sme[STATE2][RESET].nextState = IDLE ;
sme[STATE2][EVENT1].nextState = STATE2 ;
sme[STATE2][EVENT2].nextState = IDLE ; /**/
sme[STATE2][EVENT3].nextState = STATE3 ;
sme[STATE2][EVENT4].nextState = STATE2 ;
sme[STATE2][NEXT].nextState = STATE3 ;
/*State3*/
sme[STATE3][RESET].state = STATE3 ;
sme[STATE3][EVENT1].state = STATE3 ;
sme[STATE3][EVENT2].state = STATE3 ;
sme[STATE3][EVENT3].state = STATE3 ;
sme[STATE3][EVENT4].state = STATE3 ;
sme[STATE3][NEXT].state = STATE3 ;
sme[STATE3][RESET].action = reset ;
sme[STATE3][EVENT1].action = keep ;
sme[STATE3][EVENT2].action = keep ;
sme[STATE3][EVENT3].action = reset ; /**/

```

```

sme[STATE3][EVENT4].action = action4 ;
sme[STATE3][NEXT].action = action4 ;
sme[STATE3][RESET].nextState = IDLE ;
sme[STATE3][EVENT1].nextState = STATE3 ;
sme[STATE3][EVENT2].nextState = STATE3 ;
sme[STATE3][EVENT3].nextState = IDLE ; /**/
sme[STATE3][EVENT4].nextState = STATE4 ;
sme[STATE3][NEXT].nextState = STATE4 ;
/*State4*/
sme[STATE4][RESET].state = STATE4 ;
sme[STATE4][EVENT1].state = STATE4 ;
sme[STATE4][EVENT2].state = STATE4 ;
sme[STATE4][EVENT3].state = STATE4 ;
sme[STATE4][EVENT4].state = STATE4 ;
sme[STATE4][NEXT].state = STATE4 ;
sme[STATE4][RESET].action = reset ;
sme[STATE4][EVENT1].action = keep ;
sme[STATE4][EVENT2].action = keep ;
sme[STATE4][EVENT3].action = keep ;
sme[STATE4][EVENT4].action = reset ; /**/
sme[STATE4][NEXT].action = reset ;
sme[STATE4][RESET].nextState = IDLE ;
sme[STATE4][EVENT1].nextState = STATE4 ;
sme[STATE4][EVENT2].nextState = STATE4 ;
sme[STATE4][EVENT3].nextState = STATE4 ;
sme[STATE4][EVENT4].nextState = IDLE ; /**/
sme[STATE4][NEXT].nextState = IDLE ;

```

...

B1 ASCII Tabellen

Codewerte *hexadezimal*

00 nul	01 soh	02 stx	03 etx	04 eot	05 enq	06 ack	07 bel
08 bs	09 ht	0a nl	0b vt	0c np	0d cr	0e so	0f si
10 dle	11 dc1	12 dc2	13 dc3	14 dc4	15 nak	16 syn	17 etb
18 can	19 em	1a sub	1b esc	1c fs	1d gs	1e rs	1f us
20 sp	21 !	22 "	23 #	24 \$	25 %	26 &	27 '
28 (29)	2a *	2b +	2c ,	2d -	2e .	2f /
30 0	31 1	32 2	33 3	34 4	35 5	36 6	37 7
38 8	39 9	3a :	3b ;	3c <	3d =	3e >	3f ?
40 @	41 A	42 B	43 C	44 D	45 E	46 F	47 G
48 H	49 I	4a J	4b K	4c L	4d M	4e N	4f O
50 P	51 Q	52 R	53 S	54 T	55 U	56 V	57 W
58 X	59 Y	5a Z	5b [5c \	5d]	5e ^	5f _
60 `	61 a	62 b	63 c	64 d	65 e	66 f	67 g
68 h	69 i	6a j	6b k	6c l	6d m	6e n	6f o
70 p	71 q	72 r	73 s	74 t	75 u	76 v	77 w
78 x	79 y	7a z	7b {	7c	7d }	7e ~	7f del

Codewerte *dezimal*

0 nul	1 soh	2 stx	3 etx	4 eot	5 enq	6 ack	7 bel
8 bs	9 ht	10 nl	11 vt	12 np	13 cr	14 so	15 si
16 dle	17 dc1	18 dc2	19 dc3	20 dc4	21 nak	22 syn	23 etb
24 can	25 em	26 sub	27 esc	28 fs	29 gs	30 rs	31 us
32 sp	33 !	34 "	35 #	36 \$	37 %	38 &	39 '
40 (41)	42 *	43 +	44 ,	45 -	46 .	47 /
48 0	49 1	50 2	51 3	52 4	53 5	54 6	55 7
56 8	57 9	58 :	59 ;	60 <	61 =	62 >	63 ?
64 @	65 A	66 B	67 C	68 D	69 E	70 F	71 G
72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O
80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W
88 X	89 Y	90 Z	91 [92 \	93]	94 ^	95 _
96 `	97 a	98 b	99 c	100 d	101 e	102 f	103 g
104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o
112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w
120 x	121 y	122 z	123 {	124	125 }	126 ~	127 del