Sprachumfang ANSI-C

1. Operatoren, Datentypen und Ausdrücke

1.1 Inkrement- und Dekrement Operator

Der Inkrement-Operator ++ addiert 1 zu einem, der Dekrement Operator – subtrahiert 1 von einem Operanden. Die Operatoren ++ und -- können vor (Präinkrement bzw. -dekrement) und nach (Postinkrement bzw. -dekrement)dem Operanden angegeben werden.

Beim Präinkrement und -dekrement wird vor der Verwendung/Verknüpfung des Operanden inkrementiert und dekrementiert. Beim Postinkrement und -dekrement wird nach der Verwendung bzw. Verknüpfung des Operanden inkrementiert und dekrementiert.

Beispiel:

a = 5, b = 4
c = a++ + b *Ergebnis: c = 9, a = 6, b = 4*

a = 5, b = 4
c = ++a + b *Ergebnis: c = 10, a = 6, b = 4*

a = 5, b = 4
c = a++ + ++b *Ergebnis: c = 10, a = 6, b = 5*

1.2 Arithmetische Vergleiche

Die Vergleichsoperatoren auf Unterschiedlichkeit sind > (größer) < (kleiner) >= (größer gleich) und <= (kleiner gleich) und != (verschieden von). Der Vergleichsoperator auf Gleichheit ist = =.
Die Operatoren >, <, >= und <= haben einen Vorrang gegenüber den Operatoren != und = =. Das Ergebnis eines jeden arithmetischen Vergleichs ist ein logisches Ergebnis vom Wert 1 (wahr, true), wenn der Vergleich positiv ausgefallen ist oder vom Wert 0 (falsch, false), wenn der Vergleich negativ ausgefallen ist.

wichtig:
if (a = 0) führt keinen Vergleich auf den Wert 0 durch, sondern eine Zuweisung (Ergibtanweisung) des Wertes 0 an die Variable a. Wenn a der Wert 0 zugewiesen wird, dann ist eine nachfolgende if-Bedingung immer false und der folgende Kode wird nicht abgearbeitet.

Beispiel:

a = 5, b =4
a <=b *Ergebnis: false*

1.3 Logische Vergleiche

Die logischen Vergleichsoperatoren && sowie || führen die logische UND sowie ODER-Verknüpfung -wohlgemerkt nicht die binäre bzw. bitweise Verknüpfung- zweier Operanden durch. Das Ergebnis der Verknüpfung ist ein logisches Ergebnis vom Wert 1 (wahr, true), wenn der Vergleich positiv ausgefallen ist oder vom Wert 0 (falsch, false), wenn der Vergleich negativ ausgefallen ist. Unter den logischen Vergleichsoperatoren besitzt die UND-Verknüpfung eine höhere Priorität als die ODER-Verknüpfung.

Beispiel:

a = 5, b = 4
a>3 && b>2 — *Ergebnis: true*
Die Klammerung (a>3) && (b>2) kann entfallen, da der Vergleichsoperator eine höhere Priorität besitzt

a = 5, b = 4
a && b — *Ergebnis: true*

a = 5, b = 4
a>3 && b>2 || a>100 — *entspricht ((a>3) && (b>2)) || (a>100)*
Ergebnis: true

1.4 Bitweise Verknüpfungen

Die folgenden binären Operatoren führen bitweise einen Verknüpfung von Operanden durch. Sie sind in erster Linie für ganzzahlige Variablen des Datentyps char, int und unsigned int anzuwenden.

& binäre UND-Verknüpfung
| binäre ODER-Verknüpfung
^ binäre EXKLUSIVE-ODER-Verknüpfung
~ binäre Invertierung

Dabei werden die Bits an den entsprechenden Stellen der beiden zur Verknüpfung vorgesehenen Variablen bzw. Konstanten verrechnet. Hilfreich sind besonders der & und | Operator zum Ein- und Ausblenden von Bitstellen in Variablen.

Beispiel:

a = 0xC345, c = 0
c = a & 0xFF00 — *Ergebnis: c = 0xC300*

a = 0xC345, c = 0
c = a | 0xFF00 — *Ergebnis: c = 0xFF45*

a = 0xC345, c = 0
c = a^FF00 — *Ergebnis:*
c = 1100 0011 0100 0101 ^ 1111 1111 0000 0000
= 0011 1100 0100 0101 = 0x3C45

a = 0xC345, c = 0
c = ~a — *Ergebnis:*
c = ~ 1100 0011 0100 0101= 0011 1100 1011 1010
= 0x3CBA

1.5 Bitverschiebungen

Die Operatoren >> und << führen eine Bitverschiebung um n Stellen nach rechts bzw. links durch. In die entsprechenden linksseitigen oder rechtsseitigen Stellen werden 0-Werte nachgeschoben. Insofern handelt es sich nicht um die in der Assemblertechnik häufig verwendete Bitrotation.

Beispiel:

a = 0xC345, b = 0, c = 4
b = a >> c *Ergebnis: b= 0x0C34*

a = 0xC345, c = 0,
c = a <<8 *Ergebnis: c= 0x4500*

1.6 Arithmetische Verknüpfungsoperatoren

Folgende arithmetische Zuweisungsoperatoren stehen zur Verfügung:

\+ - * / %.

Weitere arithmetische Verknüpfungsoperatoren existieren nicht. Soll z.B. eine Potenz, eine Wurzel o.ä. ermittelt werden, so wird in der Regel auf eine vorhandene Funktion zurückgegriffen, deren Deklarationen sich in der Header-Datei math.h befinden. Innerhalb dieser Funktionen wird dann auf diese Standardtypen zurückgegriffen. Der Modulo-Operator % liefert den Rest einer Division zweier ganzzahliger Operanden.

Beispiel:

a = 346, b = 4, c = 0
c = a % b *Ergebnis: c = 2*

1.7 Zusammengesetzte Zuweisungsoperatoren

Wird eine Variable arithmetisch oder binär mit sich selbst verknüpft, so bietet sich die Verwendung von zusammengesetzten Zuweisungsoperatoren an, indem ***vor*** dem Gleichheitszeichen der entsprechende Verknüpfungsoperator verwendet wird.
Folgende zusammengesetzte Zuweisungen sind möglich:

arithmetische Zuweisungen: += -= *= /= %=
binäre Zuweisungen: >>= <<= &= |= ^=

Beispiel:

a = 0xC345, c = 2
a += c *Ergebnis: c= 0xC347*

a = 2, b = 2, c=0xFF
c >>= a+b *Ergebnis: c= 0x0F*

Der Inkrement- und Dekrementoperator ohne weitere Verknüpfungen benötigt kein Gleichheitszeichen, da die Operation ja nur den Quelloperanden selbst verändert. Prä- und Postinkrement und Prä- und Postdekrement liefern ohne Verknüpfung das gleiche Ergebnis.

Beispiel:

a = 0xC345
a++ *Ergebnis: a= 0xC346*
++a *Ergebnis: a= 0xC347*

1.8 Datentypen

Folgende elementare Datentypen sind vorhanden

char	ein Byte, dient in der Regel zur Aufnahme des ASCII-Kodes eines Zeichens
int	ganzzahliger vorzeichenbehafteter Wert (meistens 2 Byte) im Bereich von -32768 bis +32767
float	Gleitkommazahl mit einfacher Genauigkeit (meistens 3 Byte Mantisse, 1 Byte Exponent)
double	Gleitkommazahl mit doppelter Genauigkeit (meistens 6,5 Byte Mantisse, 1,5 Byte Exponent)

Der Datentyp int (Integer) besitzt dabei die folgenden Untertypen

short	ganzzahliger vorzeichenbehafteter Wert (meistens 1 Byte) im Bereich von -128 bis +127
long	ganzzahliger vorzeichenbehafteter Wert (meistens 4 Byte)

Des Weiteren gibt es zu jeden Integer-Typ eine entsprechenden vorzeichenlosen Typ mit der Erweiterung unsigned (unsigned short, unsigned int, unsigned long). Hier sind nur positive ganze Zahlen darstellbar. Der Wertebereich z.B. einer unsigned int Zahl erstreckt sich dann von 0 bis +65536.

1.9 Konstanten

Konstanten können während der Laufzeit des Programms ihren Wert nicht verändern. Es handelt sich um sogenannte Right-Values (Ausdrücke, die nur auf der rechten Seite einer Zuweisung stehen können).
Die Größe und die Art und Weise der Angabe der Konstante entscheiden, ob der Compiler diese als Festkomma-, Gleitkomma- oder Textkonstante auffasst. Eine vorteilhafte Möglichkeit, mit Konstanten zu arbeiten, ist die Verwendung der Präcompilerbefehls define. Hier wird einem symbolischen Bezeichner ein Wert zugewiesen.

Beispiel:

# define NULL	*‘0’*	*Zeichen-Textkonstante*
# define LOGO	*“BC – GmbH“*	*Zeichenkette-Textkonstante*
# define PI	*3.14*	*Gleitkommakonstante*
# define ENDE	*3424*	*Integer-Festkommakonstante*

Durch den reservierten Bezeichner const in der Typvereinbarung für Variablen können diese ebenfalls als Konstanten festgelegt werden.

Beispiel:

const float PI = 3.14;

1.10 Vorrang und Reihenfolge der Bewertungen der Operatoren

Es empfiehlt sich, bei Unklarheiten über den Vorrang der Bewertungen der Operatoren eine Klammersetzung zu verwenden. DIE ÖFFNENDEN UND SCHLIEßENDEN KLAMMERN HABEN DEN HÖCHSTEN RANG ALLER OPERATOREN[1].

[1] *Schreiner, A.T.; Janich, E.* (Hrsg.): Kernighan Ritchie Programmieren in C. 2. Aufl. Leipzig: Fachbuchverlag 1990.

() [] ->
! ~ ++ -- - (Vorzeichen) * (Auflösungsoperator) &(Adressoperator)
* (Multiplikation) / %
+ -
<< >>
< <= > >=
= = !=
&
^
|
&&
||
?:
= += -= *= /= ...

1.11 if-else-Bedingung

Die if-else-Bedingung prüft einen Ausdruck (expression) auf den Wert true oder false. Die

else-Anweisung ist dabei optional.

```
if (expression)        statement1
else                   statement 2
```

Liefert expression einen Wert verschieden von 0, so ist die if-Bedingung erfüllt und es wird statement 1 ausgeführt. Existiert ein else-Zweig und die if-Bedingung liefert einen Wert 0, so wird statement 2 ausgeführt.
Oftmals ist es notwendig, längere „if-else-Ketten“ zu realisieren. Trifft die erste if-Bedingung zu, so wird statement 1 ausgeführt. Trifft sie nicht zu, so kommt eine weitere if-Bedingung mit einem statement 2, die wiederum in ihrem else-Zweig eine weitere if-Bedingung besitzt usw. Die letzte else-Anweisung berücksichtigt dann den Fall, dass gar keine Bedingung erfüllt ist.

Hinweis:
Bei verschachtelten if-else-Anweisungen ist konsequent darauf zu achten, dass ein eventuell auftretender else-Teil immer zur letzten if-Bedingung gehört. Gegebenenfalls ist durch entsprechende Klammern die optionale else-Anweisung der entsprechenden if-Bedingung zuzuordnen. Das Einrücken der entsprechenden else-Anweisung auf die Höhe der if-Anweisung ist nur eine Orientierung für den Programmierer.

Beispiel:

```
if (a<b)
  {…}                  wenn a<b
else
  {  }                 wenn a>=b

if (34)                Bedingung ist immer erfüllt

if (a>2)
   if (a>4)
      b= 44;
```

else b=43;	*else-Zweig zur if-Bedingung (a>4) - trotz Einrückens*
if (a=4)	*liefert immer den Wert true, da in der if-Bedingung eine Zuweisung und keine Abfrage steht, richtig wäre if (i ==4)*

Bei einfachen if-else-Anweisungen in Verbindung mit einer Zuweisung besitzt C eine spezielle Syntax:
wert = (expression?wert1:wert2)

Wenn expression mit 1 bewertet wird, dann erfolgt eine Zuweisung von wert1 an die Variable wert, wenn expression mit 0 bewertet wird, erfolgt eine Zuweisung von wert2 an wert.

Beispiel:

a=(a>=grenz?grenz:a+1)	*wenn a größer oder gleich grenz ist, dann wird a der Wert von grenz zugewiesen, wenn nicht erfolgt ein Inkrementieren von a*

1.12 while-Schleifen

Bei einer while-Schleife wird die Schleifenbedingung vor dem Eintritt in die Schleife getestet. Es handelt sich um eine abweisende Schleife. Liefert expression einen Wert verschieden von 0, so ist die Schleifenbedingung erfüllt und die Schleife bzw. statement wird abgearbeitet.

```
while (expression)
   statement
```

Nach dem Verlassen der Schleife wird erneut die Bedingung bzw. expression getestet. Innerhalb der Schleife sollte deshalb softwaremäßig auf die Schleifenbedingung Einfluss genommen werden. Unter Umständen kann das Ergebnis der Schleifenbedingung auch unabhängig von der Software durch die Hardware beeinflusst werden.

Beispiel:

while (!((c=getchar())=='a')) {...}	*die Schleife wird so lange durchlaufen, bis das Zeichen a von der Standardeingabe eingegeben wird, erst dann wird expression mit 0 bewertet und die Schleife verlassen*
while (a<b) {...}	*die Schleife wird so lange abgearbeitet, wie a < b ist*
while (a&0X0001) {...}	*die Schleife wird so lange abgearbeitet, wie das niederwertigste Bit in a gesetzt ist*
while(1) {...}	*Realisierung einer Endlosschleife*

1.13 do-while-Schleifen

Oftmals soll erst am Ende der Schleife die Abbruchbedingung getestet werden (nichtabweisende Schleife). Bei der do-while-Schleife wird statement nach do ausgeführt und am Ende der Schleife expression getestet. Diese Schleife wird also mindestens einmal ausgeführt.

do
  statement
while (expression)

Beispiel:

do
  {...}
while (a<b); — *Verlassen der Schleife, wenn die Bewertung a<b den Wert 0 liefert, also wenn a>=b ist*

1.14 for-Schleifen

Bei einer for-Anweisung kann gleichzeitig eine Anfangsbedingung, eine Abbruchbedingung und eine Aktion in einem einzigen Befehl untergebracht werden.

for(expression1; expression2; expression3)
Statement

Expression1 und expression3 sind Zuweisungen und expression2 ist eine Bedingung. Alle drei expressions sind optional. Die expressions sollten, wenn möglich alle einen Bezug zur Schleife besitzen bzw. mit der Kontrolle der Schleife in Verbindung stehen. Bei der Abarbeitung der for-Schleife wird zuerst die expression1 einmalig abgearbeitet, dieses ist die Startzuweisung. Am Anfang der Schleife wird expression 2 bewertet und entschieden, ob die Schleife abgearbeitet wird. Es handelt sich hier wie bei der while-Schleife ebenfalls um eine abweisende Schleife. Liefert expression 2 einen Wert verschieden von Null, so wird die Schleife erneut abgearbeitet und am Ende expression 3 ausgeführt. Danach findet ein erneutes Bewerten von expression2 statt. Fehlt expression 2, so wird diese als wahr angenommen. Fehlen expression 1 und expression 3, so werden keine einmalige Startzuweisung und auch keine Zuweisungen am Ende eines Schleifendurchlaufs abgearbeitet. Die for-Schleife bietet sich besonders bei Zählschleifen mit einer festen Anzahl von Durchläufen an.

Beispiel:

for(i=0; i<=10 ;i++) {…} — *Setzen der Anfangsbedingung i=0; Prüfen zu Beginn eines jeden Durchlaufes, ob i kleiner oder gleich 10 ist, wenn dieses der Fall ist, Inkrementieren von i und erneutes Abarbeiten der Schleife, Verlassen der Schleife, wenn die Bewertung a<=10 den Wert 0 liefert bzw. wenn a den Wert 11 erreicht hat – Durchführung von insgesamt 11 Schleifendurchläufen*

for(;;) — *Abarbeiten einer Endlosschleife*

2. Zeiger, Felder, Strukturen, Varianten

2.1 Zeiger und Adressen

Mit Zeigern kann indirekt, über die entsprechende Speicheradresse auf Objekte zugegriffen werden. Zeiger- oder Pointervariablen enthalten die Startadresse eines Objektes (in der Regel die einer Variable) im Speicher.
Zeiger- oder Pointervariablen besitzen einen Typ. Dieser Pointertyp richtet sich nach dem Typ der Objekte, dessen Startadresse die Zeigervariable besitzen kann. Für einen Variablentyp int existiert somit ein Zeiger vom Typ int. Zeiger werden mithilfe des *-Operators definiert.

Beispiel:

*int wert,*pt;*	*Definition einer Variable wert vom Typ int und eines Zeigers pt vom Typ int*
*char zeichen='a', *ptr=0X2000;*	*Definition einer Variable vom Typ char und Initialisierung mit 'a' und eines Zeigers ptr und Initialisierung mit 0X2000*

wichtig: Nach dem Definieren eines Zeigers besitzt dieser noch keinen Wert und zeigt auf einen undefinierten Ort, ins „Nichts". Nichtinitialisierte Zeiger müssen vor Verwendung einen Wert zugewiesen bekommen. Nichtinitialisierte Zeiger sind häufige Fehlerursache in C-Programmen.

Soll nun die Adresse eines Objektes ermittelt werden und diese z.B. einem Zeiger zugeordnet werden, so wird der &-Operator als Vorzeichen-Operator des Objektes verwendet, dessen Adresse ermittelt werden soll.

Beispiel:

*int wert,*pt;*	
pt = &wert;	*Der Adressvariable pt wird die Adresse von wert zugewiesen.*

Besitzt eine Adressvariable einen gültigen Wert, so kann mit Hilfe des *-Operators (Auflösungsoperator) die Adresse aufgelöst und auf das adressierte Objekt zugegriffen werden.

Beispiel:

*int wert1 = 7,wert2,*pt;*	
pt = &wert1;	*Der Adressvariable pt wird die Adresse von wert1 zugewiesen.*
*wert2=*pt;*	*Der Variable wert2 wird der Inhalt der durch pt adressierten Integervariable zugewiesen, wert2 ist danach 7.*

Das gleiche Ergebnis wird auch durch eine Direktzuweisung von wert1 an wert2 erreicht

wert2 = wert1;

wichtig: Der *-Operator besitzt in Verbindung mit Zeigern eine Doppelfunktion. So charakterisiert er in der Definitionszeile einer Variable diese als Zeiger. In einer Befehls- oder Anweisungszeile löst er die entsprechende Adressvariable auf.

Ein Zeiger kann immer nur auf ein Objekt des Typs zeigen, für den er definiert wurde. Einer Zeigervariable z.B. vom Typ int kann nicht zur Adressierung einer char-Variable verwendet werden. Eine Zeigervariable beinhaltet immer die Startadresse eines Objektes für dessen Typ sie vorgesehen ist, so dass eine Vermischung der Typen im ersten Moment möglich erscheint, da es sich in allen Fällen immer um die Startadresse von Objekten handelt. Wegen der nachfolgend erläuterten sehr leistungsfähigen Zeigerarithmetik können diese aber nur auf Objekte entsprechend ihres vereinbarten Typs zeigen.

2.2 Zeiger als Argumente in Funktionsaufrufen

Beim Aufruf von Funktionen/Unterprogrammen können an diese bei Bedarf Werte übergeben werden. Diese Kopien werden entweder in Prozessorregistern oder über den Stack an das Unterprogramm übergeben. Im Unterprogramm wird diese Kopie der übergebenen Variablen mit temporären Variablennamen versehen und entsprechend verarbeitet. Die Originalvariablen können auf diesen Weg nicht beeinflusst werden.

Beispiel:

```
void austausch (int wert1, int wert2)      Definition eines Unterprogramms zum Austausch
  {                                        zweier Variablen
  int dummy;
  dummy = wert1; wert1 = wert2; wert2 = dummy;
  }

int a=4,b=5;
...
austausch(a,b)                             Es werden nicht die Variablen a und b ausge-
                                           tauscht, sondern nur Kopien der Variablen a und
                                           b, es gilt nach Beendigung des Unterprogramms
                                           weiterhin a=4 und b=5.
```

Eine Abhilfe schafft hier die Übergabe von Zeigern. Im Hauptprogramm werden die Zeiger entsprechend der Variablen initialisiert, die vom Unterprogramm verändert werden.

Die konsequente Werteübergabe scheint im ersten Moment die Flexibilität von Unterprogrammen einzuschränken. Sie führt jedoch zu einer Vereinfachung von Unterprogrammaufrufen und erweist sich gegenüber der Variablenübergabe als bedeutend leistungsfähiger.

Beispiel:

```
void austausch (int *wert1, int *wert2)    Definition eines Unterprogramms zum Austausch
                                           zweier Variablen, es werden zwei Zeiger
                                           übergeben
  {
  int dummy;
  dummy = *wert1;
  *wert1 = *wert2;
```

```
  *wert2 = dummy;
  }

int a=4,b=5;
...
austausch(&a,&b);
```

Es werden die Adressen der Variablen a und b übergeben und im Unterprogramm die Inhalte der Adressen ausgetauscht, am Ende gilt a=5 und b=4

2.3 Zeigerarithmetik

Mit Zeigern lassen sich zahlreiche Rechenoperationen durchführen. So können Pointer in- und dekrementiert werden. Hierbei zeigt im Ergebnis des In- bzw. Dekrementierens der Pointer auf das nachfolgende bzw. das vorhergehende Element der Folge von Objekten des Typs, den der Zeiger besitzt. Der Zahlenwert des Zeigers wird also um die Byteanzahl erhöht bzw. vermindert, die ein Element des Variablentyps benötigt, für den der Zeiger vereinbart wurde. Das Inkrementieren eines int-Pointers führt z.B. bei den meisten Mikroprozessoren zu einer Addition des Wertes 2 zum ursprünglichen Zeigerwert, vorausgesetzt eine Integer-Variable benötigt zwei Byte Speicherplatz.

Beispiel:

```
int wert1,wert2,*pt;
pt = &wert1;
```

Der Adressvariable pt wird die Adresse von wert1 zugewiesen

```
pt++;
```

pt wird inkrementiert und zeigt auf die nächste Integervariable im Speicher bzw. auf wert2, wobei der Wert von pt um 2 erhöht wurde, insofern der Typ int 2 Byte im Speicher benötigt wenn wert1 und wert2 im Speicher zusammenhängend nacheinander untergebracht sind

Zu Zeigervariablen können ganzzahlige Werte hinzuaddiert und subtrahiert werden. Eine Operation p+i ändert den Wert von p in der Art, dass p nach der Operation auf eine Variable zeigt, die genau i Elemente von der Stelle auf die p bisher verwiesen hat in Richtung aufsteigender Adressen entfernt ist.

Beispiel:

```
int i=3,wert1,wert2, wert3,wert4,*pt;
pt = &wert1;
```

Der Adressvariable pt wird die Adresse von wert1 zugewiesen

```
pt += i;
```

pt wird um 3 Elemente erhöht und zeigt dann auf die Variable wert4, der Zahlenwert von pt wird um 6 erhöht, wenn der Typ int 2 Byte im Speicher benötigt

Zeiger können subtrahiert werden. Das Ergebnis ist ein ganzzahliger Wert, der die Distanz zwischen den Variablen angibt, auf welche die beiden subtrahierten Zeiger verweisen, bzw. die Anzahl der Elemente liefert, die sich zwischen den beiden Verweisen der subtrahierten

Zeiger befindet. Eine Addition, Multiplikation und Division von Zeigervariablen ist nicht erlaubt.
Ebenfalls können mit Zeigern Vergleiche auf Gleichheit (= =) und Ungleichheit (!=, <=, >=, >,<) durchgeführt werden.
Es ist auch möglich, den Typ von Zeigern mithilfe des cast-Operators umzuwandeln.

Beispiel:

```
int *pt1;
char *pt2 ;

pt2 = (char *) pt1;
```

Bei der Verwendung des *-Auflösungsoperators wird in einem Ausdruck *(pt+i) die Variable adressiert, welche sich genau i Elemente des Typs für den der Zeiger definiert wurde in Richtung aufsteigender Adressen im Speicher befindet. Wegen des höheren Rangs des *Auflösungsoperators ist hier unbedingt eine Klammer zu setzen,. Der Ausdruck *pt+i löst erst den Zeiger auf und addiert dann zum Zeigerinhalt den Wert i.

2.4 Strukturen

Strukturen sind Zusammenfassungen/Ansammlungen von einer oder mehreren Variablen unterschiedlichen Typs unter einem gemeinsamen Bezeichner. Die Komponenten einer Struktur sind Variablen mit einem eigenen Variablennamen. Strukturen können als Komponenten ebenfalls wiederum Strukturen enthalten. Bevor Strukturvariablen definiert werden, ist eine Deklaration mit dem struct-Befehl erforderlich.

Beispiel:

```
struct adr                          Deklaration eines Strukturtyps adr
  {
  int wert1;
  int wert2;
  };

struct adr adresse1, adresse2;      Definition von zwei Strukturvariablen adresse1
                                    und adresse2
```

Deklaration und Definition können auch in einer Befehlszeile erfolgen. Eine Initialisierung der Komponenten einer Strukturvariable in der Definitionszeile ist ebenfalls möglich. Bei einer Initialisierung werden die Anfangswerte in der Reihenfolge der Komponenten der Struktur in geschweiften Klammern aufgelistet.

Beispiel:

```
struct adr                          Deklaration eines Strukturtyps adr
  {
  int wert1;
  int wert2;
  };
```

struct adr adresse1, adresse2={1236, 4287};	*Definition von zwei Strukturvariablen adresse1 und adresse2 vom Typ adr und Initialisierung der Komponenten von adresse2*

Auf die Komponenten einer Struktur wird mithilfe des .-Operators zugegriffen. Dieser befindet sich zwischen dem Namen der Strukturvariable und dem Namen der Komponente.

Beispiel:

```
struct adr
  {
  int wert1;
  int wert2;
  } adresse1;

adresse1.wert1 = 1236;
```

Wenn innerhalb einer Struktur einer weitere Struktur auftaucht, kann der .-Operator mehrfach auftreten.
Nicht erlaubt ist die Verwendung der gleichen Struktur als Komponente. Eine Struktur muss vor ihrer Verwendung dem Compiler vollständig bekannt sein. Wird diese aber schon vor der geschweiften schließenden Klammer verwendet, so ist der vollständige Aufbau der Struktur noch unbekannt und es wird eine Fehlermeldung generiert. Außerdem würden solche Strukturen, die sich selbst als Komponente beinhalten, einen Speicherüberlauf verursachen. Eine Struktur darf sich also nicht selbst als Komponente besitzen.
Denkbar sind jedoch Zeigervariablen auf einen Strukturtyp, welcher der Struktur selbst entspricht. In einem solchen Fall wird von einer rekursiven Struktur gesprochen.

Beispiel:

```
struct adr
  {
  int wert1;
  int wert2;
  struct adr *ptr;
  };
```

2.5 Zeiger auf Strukturen

Zeiger können ebenfalls auf Strukturen verweisen. Dem Zeiger ist auch hier, wie bei den Standardtypen char, int und float, der Typ der Struktur zuzuweisen, auf die der Pointer verweist. Ebenso können Felder von Strukturen vereinbart werden.

Beispiel:

```
struct adr                          Deklaration eines Strukturtyps adr
  {
  int wert1;
  int wert2;
  };
```

*struct adr adresse[2],*point;*	*Definition eines Feldes von Strukturen des Typs adr und eines Zeigers ptr auf den Typ adr*

Auch gelten die im Abschnitt Zeigerarithmetik gemachten Aussagen hinsichtlich des In- und Dekrementierens sowie des Addierens und Subtrahierens mit einem ganzzahligen Wert. Wird ein Zeiger mithilfe des &-Operators initialisiert, so verweist er auf die Adresse der ersten Komponente der verwendeten Strukturvariable im Speicher. Auf die Elemente einer Struktur wir mit dem ->-Operator (Auswahl-Operator) zugegriffen. Vor dem ->-Operator steht der Name der Zeigervariable und danach die Komponente.

Beispiel:

struct adr *{* *int wert1;* *int wert2;* *};*	*Deklaration eines Strukturtyps adr*
int i;	
*struct adr adresse1,*point;*	*Definition einer Variable adresse1 des Typs adr und eines Zeigers point auf den Typ adr*
point = &adresse1;	*Initialisierung der Zeigervariable*
i = point->wert1;	*Zugriff auf eine Komponente der Struktur*

Eine weitere Möglichkeit, auf Komponenten einer Struktur zuzugreifen, ist die Verwendung des bereits bekannten *-Auflösungsoperators. Die Syntax i = (*point).wert1 würde das gleiche Ergebnis wie im Beispiel vorab liefern. Auch ist der Vorrang des .-Operators durch eine entsprechende Klammerung zu unterdrücken.

2.6 Vektoren/Felder

Bei Vektoren bzw. Feldern wird eine feststehende Anzahl typgleicher Variablen mit einem gemeinsamen Bezeichner versehen. Jedes Feldelement besitzt also einen Feldnamen und eine Feldnummer. Felder werden durch das Benennen des Feldnamens hinter einem Typbezeichner und der Anzahl der Feldelemente in eckiger Klammer definiert.

Beispiel:

int wert[10];	*Definition eines Feldes wert mit 10 Elementen vom Typ int*
*char *zeichen[10];*	*Definition eines Feldes zeichen mit 10 Zeigern auf char*

Die Elemente werden im Speicher in aufsteigender Reihenfolge zusammenhängend untergebracht. Es ist möglich, bei der Definition des Vektors diesen mit Anfangswerten zu versehen. Auch können mehrdimensionale Vektoren definiert werden. Allerdings muss beachtet werden, dass der Speicherplatzbedarf mit wachsender Feldgröße sehr stark ansteigt. Bei zweidimensionalen Feldern stellt der erste Wert die Anzahl der Spalten und der zweite Wert die Anzahl der Zeilen der so aufgebauten zweidimensionalen Matrix dar.

Beispiel:

int wert[4] = {10, 223, 1746, –297};	*Definition und Initialisierung eines int-Feldes wert mit 4 Elementen*
*char *zeichen[2][2] = {* *{0X012F, 0X78C9},* *..{0X0400, 0X442F}* *}*	*Definition und Initialisierung eines Feldes von char-Zeigern mit 2 mal 2 Elementen*

Der Zugriff auf Feldvariablen erfolgt mithilfe ihres Feldnamens mit nachfolgender Feldnummer.

Beispiel:

```
int wert[4],*zeichen[2][2];
wert[0] = 0;   wert[1] = 1;   wert[2] = 2;   wert[3] = 3;
zeichen[0][0] = &wert[0];   zeichen[0][1] = &wert[1];
zeichen[1][0] = &wert[2];   zeichen[1][1] = &wert[3];
```

wichtig: Es ist zu beachten, dass ein Feld mit n Elementen die Feldnummern 0, 1, 2, ... (n-1) besitzt. Die Aufzählung beginnt bei der Feldnummer 0 und endet bei (n-1). Die Verwendung einer Feldnummer mit dem Wert n oder eines Zahlenwerte größer als n führt zwar zu keinem Fehler des Compilers, es wird aber auf einen Variablenbereich außerhalb des Feldes zugegriffen.

Da die Feldnummer vom Compiler nicht überwacht wird, kann diese bei einer Felddefinition mit gleichzeitiger Initialisierung auch weggelassen werden. Der Übersetzer zählt die Initialisierungen, führt die Wertzuweisungen durch und dimensioniert den Vektor entsprechend. Bei einer solchen Definition befindet sich keine Nummer zwischen den eckigen Klammern.

Beispiel:

int wert[] = {10, 223, 1746, –297};	*Definition und Initialisierung eines Feldes wert mit 4 Elementen und int*

Die Verwendung des Feldbezeichners ohne eckige Klammern und Feldnummer stellt in der Programmiersprache C eine Zeigerkonstante zur Verfügung, die auf das erste Element des benannten Feldes zeigt. Diese Adresskonstante kann einem Zeiger des gleichen Typs zugewiesen werden. Hinsichtlich der Arithmetik mit der Zeigerkonstante gelten die gleichen Regeln wie im Abschnitt Zeigerarithmetik.

wichtig: Es ist allerdings zu berücksichtigen, dass der Feldname eine Adresskonstante liefert, die nicht verändert werden kann (Right-Value).

Beispiel:

*int wert[4],*pt1,*pt2,*pt3;*	*Definition und Initialisierung eines int-Feldes wert mit 4 Elementen und von 3 Zeigern auf int*
pt1 = wert;	*Zuweisen der Adresse des ersten Elements des Feldes wert an pt1*

pt2 = wert+1;	*Zuweisen der Adresse des zweiten Elements des Feldes wert an pt2*
pt3 = ++wert;	*nicht erlaubt, wert ist eine Adresskonstante und kann nicht inkrementiert werden*

2.7 Gemeinsamkeiten und Unterschiede bei Feldern und Zeigern

Da mit dem Feldnamen ein Zeiger bereitgestellt wird, der die Adresse des ersten Elements des Feldes beinhaltet und die bekannte Zeigerarithmetik auch uneingeschränkt bei Feldern angewendet werden kann, bestehen viele Gemeinsamkeiten zwischen Pointern und Feldern. Jede Operation mit Feldern kann auch mit Pointern durchgeführt werden. Später wird gezeigt, dass der Umkehrung dieser Aussage - Jede Operation mit Pointern kann auch mit Feldern durchgeführt werden - nicht uneingeschränkt möglich ist. Die wichtigste Gemeinsamkeit von Pointern und Feldern ist, dass ein Zeiger auf einen Variablentyp bereitgestellt wird. Bei Feldern wird ein Zeiger auf einen Vektor dieses Typs und bei Pointern auf eine Variable dieses Typs generiert.
Wird ein Zeiger definiert, so wird nur der Speicherplatz für die Zeigervariable reserviert. Bei einem Feld hingegen wird Speicherplatz für jedes Feldelement bereitgestellt.

Beispiel:

char feld[100];	*Reservieren von 100 Byte Speicherplatz*
*char *pt;*	*Reservieren von Speicherplatz für eine Adressvariable auf den Typ char (in der Regel 2 Byte)*

Es gelten folgende Zusammenhänge:

feld[i] = *(feld+i)
feld = &feld[0]
feld+i = &feld[i]

Wegen dieser Zusammenhänge ist es sinnvoll, einen Zeiger nicht nur mit Hilfe seiner Adressvariable und einer Distanz aufzulösen, sondern auch mit seiner Adressvariable und einem Indexwert und es gilt:

*(pt+i) = pt[i]

Es wird noch einmal hervorgehoben, dass es sich bei dem Feldnamen um eine Adresskonstante handelt, deren Wert immer unverändert bleibt. Ein Ausdruck feld = pt im Beispiel vorab weist der Adresskonstante Feld den Wert der Adressvariable pt zu und ist deshalb nicht zulässig. Umgekehrt kann aber der Wert der Adresskonstante einem Zeiger zugewiesen werden pt = feld. Die Adresskonstante besitzt einen festen und unveränderlichen Wert. Die Zeigervariablen hingegen muss initialisiert werden und kann zur Programmlaufzeit neue Werte annehmen.

Hinweis: Bei der Übergabe von Vektoren an Unterprogramme wird die Adresse (Adresskonstante) des ersten Elements übergeben. Innerhalb der aufgerufenen Funktion wird diese übergebene Adresskonstante in eine lokale Variable kopiert, so dass dann wiederum Änderungen dieses Adresswertes möglich sind

2.8 Varianten

Besonders bei der hardwarenahen Programmierung ist es oftmals erforderlich, auf ein und denselben Speicherbereich Zugriffe mittels verschiedener Datentypen durchzuführen. Soll

z.B. der Wert einer Single-Precision-Floating-Point-Variable (Datentyp float - 4 Byte) über eine serielle Schnittstelle übertragen werden, so ist ein byteweiser Zugriff auf den Speicherbereich dieser Variable erforderlich (Daten werden über eine RS 232-Schnittstelle immer byteseriell übertragen). Mithilfe von Zeigern lässt sich eine solche Aufgabe realisieren. Eine günstigere Möglichkeit stellen aber Varianten dar, welche Zugriffe auf die Floating-Point-Zahl mit einem 4-Byte-Zugriff über den Datentyp float und ebenfalls durch vier Bytezugriffe vom Datentyp char zulassen. Eine Variante wird mit dem reservierten Bezeichner union deklariert und definiert. Auf die Komponenten wird mittels des **.**-Operators (genau wie bei den Strukturen) zugegriffen.

Beispiel:

```
union vari                          Deklaration einer Variante vom Typ vari mit der
  {                                 Möglichkeit eines float- und char-Zugriffs
  float zahl_float;
  char feld[4];
  };

union vari wert1,wert2;             Definieren zweier Varianten wert1 und wert2 vom
                                    Typ vari

....

wert1.zahl_float = 3.14;            Zugriff auf die Variante über den Datentyp float
for (i=0;i<4;i++)                   Zugriff auf die Variante über den Datentyp char
  send = wert1.feld[i];             (byteweise)
```

Bei der Deklaration von Bitfeldern finden Varianten ebenfalls oftmals Anwendung. Als erstes wird eine Struktur deklariert, deren Elemente einzelne Bits (Anzahl: 8, 16, 24, ...) sind. Auf diese Elemente der Struktur kann nun bitweise zugegriffen werden. In einer nachfolgend deklarierten Variante ist es dann möglich, byteweise über den Datentyp char oder wortweise über den Datentyp int auf Gruppen dieser Bitfelder geschlossen zuzugreifen.

Beispiel:

```
struct bitfeld                      Deklaration einer Struktur vom Typ bitfeld
  {                                 mit mehreren Bit-Komponenten
  unsigned bit_0 :1;
  unsigned bit_1 :1;
  unsigned bit_2 :1;
  unsigned bit_3 :1;
  unsigned dummy : 4;
  } bits_ out;                      Definition einer Strukturvariable bits_out vom Typ
                                    bitfeld zusammen mit der Deklaration

union vario                         Deklaration einer Variante vom Typ vario mit
  {                                 einem Datentyp bitfeld und einem Datentyp char
  struct bitfeld ausgangsbits;
  char wert_c;
  } flags_out;                      Definition einer Variante flags_out vom Typ vario
...
```

```
bits_out.bit_0 = 1; bits_out.bit_1 = 0;    Beschreiben der Struktur
bits_out.bit_2 = 1; bits_out.bit_3 = 1;

flags_out. ausgangsbits = bits_out;        Zuweisen der Struktur an die Variante
...
flags_out.wert_c &= 0X0F;                  Zugriff auf die Variante über den Datentyp char
...
```

3. Ein-Ausgabe-Operationen

Werden Ein-Ausgabeoperationen unter Zuhilfenahme bestehender Funktionen in der Bibliothek stdio.lib der Programmierumgebung durchgeführt, so ist die entsprechende Include-Datei stdio.h am Beginn des Quellprogramms einzubinden. Die wichtigsten Funktionen für das Bedienen der Standardein- und Ausgabe sind:

```
char getkey (void);
char getchar (void);
char putchar (char);
int printf   (const char *, ...);
int sprintf  (char *, const char *, ...);
int scanf (const char *, ...);
int sscanf (char *, const char *, ...);
```

Beispiel:

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

3.1 Standardausgaben

Mit dem putchar-Befehl bzw. der Funktion char putchar (char wert) wird das Zeichen wert zur Standardausgabe übertragen. In Mikrocontrollersystemen ist dieses in der Regel die serielle Schnittstelle zum Host-PC. Dieser stellt dann die an der COM-Schnittstelle eingehenden Informationen mithilfe einer Terminalsoftware auf dem Monitor dar. Der Rückgabewert der Funktion vom Typ char enthält Statusinformationen der Ausgabe und wird meistens nicht weiter ausgewertet.

Beispiel:

putchar('A');	*Ausgabe des Zeichens A*
putchar(0X30);	*Ausgabe des ASCII-Zeichens mit der Nummer 0X30 entspricht dem Zeichen "0"*

Mit dem printf-Befehl werden unter anderem numerische Werte in eine Zeichenkette umgewandelt und zur Standardausgabe übertragen.

printf("control", arg1, arg2, ...)

Dabei werden die Argumente arg1, arg2, ... entsprechend der Zeichenkette control formatiert. Control kann dabei nicht nur Formatierungsoperatoren, sondern auch gewöhnliche Zeichen beinhalten. Diese werden unverändert ausgegeben.

Beispiel:

printf("Stromistwert:")

Wie bereits vorab erwähnt, kann control Formatierungsanweisungen beinhalten, welche die Ausgabe der Argumente arg1, arg2, ... steuern. Jede Formatierungsanweisung beginnt mit dem %-Zeichen. Nachfolgend sind die wichtigsten Formatierungsparameter aufgelistet:

%d dezimale Darstellung
%x hexadezimale Darstellung
%u vorzeichenlose Darstellung (dezimal)
%c Ausgabe des numerischen Wertes als ASCII-Zeichen
%s Ausgabe einer Zeichenkette (Textstring)
%e Ausgabe eine float oder double Zahl in Exponentialdarstellung
%f Ausgabe einer Float-Zahl in Dezimaldarstellung

Bei mehreren Argumenten ist jedem dieser eine Formatierungsanweisung zuzuordnen! Durch die Angabe eines numerischen Wertes kann zusätzlich die Anzahl der auszugebenden Stellen des Arguments festgelegt werden. Bei Float-Zahlen können durch einen Punkt getrennt die Vor- und Nachkommastellen festgelegt werden.

Beispiel:

```
float i_ist, u_ist;
unsigned int anz;
...

printf("Stromistwert: %2.2f   Spannungsistwert: %2.2f   Messreihe: %u", i_ist, u_ist, anz);
```

Hinweis: Bei der Verwendung des printf-Befehls ist unbedingt darauf zu achten, dass die Anzahl der Platzhalter/Formatierungsanweisungen mit der Anzahl und dem Typ der folgenden Argumente übereinstimmt. Eine Überprüfung der Anzahl und des Typs der Argumente wird vom Compiler nicht durchgeführt.

Reservierte Operatoren sind \n (carrige return line-feed - Zeilenumbruch und Rücksprung an den Anfang der neuen Zeile) und \r (line feed - Rücksprung an den Anfang der bestehenden Zeile).

Beispiel:

```
float i_ist, u_ist;
unsigned int anz;
...

printf("Stromistwert: %2.2f   Spannungsistwert: %2.2f   Messreihe: %u  \r", i_ist, u_ist, anz);
...
printf("Programm abgebrochen \n Neustart erforderlich !!!")
```

3.2. Standardeingaben

Mit dem getkey-Befehl wird ein Zeichen von der Standardeingabe eingelesen. Ist keine Eingabe erfolgt, so wird ein vorher vereinbarter Wert zurückgeliefert. Im Gegensatz zu getkey wartet getchar so lange, bis ein Wert eingegeben wurde und der ASCII-Kode des entsprechenden Zeichens wird von der Funktion zurückgeliefert.

Beispiel:

putchar(getchar())	*Einlesen eines Zeichens von der Standardeingabe und Übergabe an die putchar-Funktion, die diesen Wert zur Standardausgabe leitet*

...

Mit dem scanf-Befehl werden Zeichen von der Standardeingabe gelesen und mithilfe einer Formatanweisung an Argumente übertragen. Die allgemeine Syntax lautet:

scanf(“control”, arg1, arg2,...).

Die Zeichenkette control enthält unter Umständen die auch bei printf bereits vorgestellten Formatanweisungen.

Beispiel:

int i;
char zeichen;
...

scanf(“%d %c“,&i, &zeichen);	*die eingegebene Zeichenfolge (z.B. „4321 a“) wird in eine Integer-variable i und eine Char-Variable c formatiert*

Zwischen den eingegebenen Argumenten (im Beispiel 4321 und a) kann eine beliebige Anzahl von Freizeichen stehen, diese werden ausgesondert. Zu beachten ist, dass die Argumente des scanf-Befehls Zeiger sind. Wird mit Variablennamen gearbeitet, so ist an die scanf-Funktion mithilfe des &-Operators der Adresswert der Variable zu übergeben.

3.3 Ein- und Ausgaben mit dem Speicher

Mit sprintf wird eine Ausgabe in einen String in den Speicher vorgenommen und mit sscanf von einem String aus dem Speicher eingelesen. Die Syntax dieser Befehle lautet:

sprintf(string, “control”, arg1, arg2, ...).
sscanf(string, “control”, arg1, arg2, ...).

Das Argument string dieser Funktionen ist ein Zeiger auf einen Speicherbereich, an dem der vorab bereitgestellte (sscanf) oder der im Ergebnis des Funktionsaufrufs bereitzustellende (sprintf) Textstring steht. Mit der Formatierungsanweisungen control werden die Argumente arg1, arg2, ... wie bei printf und scanf aufbereitet.
Sprintf erzeugt einen formatierten Zeichenstring im Speicher, sscanf überträgt einen Zeichenstring aus dem Speicher in entsprechende Variablen. Im Gegensatz zu printf ist das Ziel nicht die Standardausgabe, sondern ein (ausreichend großer) String im Speicher nimmt die Zeichenkette auf. Bei sscanf ist die Quelle nicht die Standardeingabe, sondern eine vorhandene Zeichenkette im Speicher.
Sprintf und sscanf nehmen dem Programmierer das aufwendige Formatieren von Zeichenketten für Ein- und Ausgabeoperationen ab. Nachdem für die Ausgabe eine fertige Zeichenkette erzeugt wurde, kann diese nun entsprechend der zur Verfügung stehenden Hardware einfach ausgegeben werden. Genau umgekehrt verhält es sich bei der Eingabe. Eine

eingelesenen Zeichenkette wird im Speicher abgelegt und dann mit sscanf-Befehl entsprechend der control-Anweisung ausgewertet.

Beispiel:

```
char buffer[40];
int wert=1234, ausg;
unsigned char xdata ausgabe _at_ 0XFD00;
...

sprintf(buffer,"Das ist ein Textsstring mit der Hexzahl  %x",wert);
for (i=0;i<40;i++)
   ausgabe =buffer[i];          spezielles Ausgaberegister auf der Adresse 0XFD00
...
printf("%s",buffer);            Ausgabe des zuvor aufbereiteten und in der Variable
                                buffer abgelegten Textstringes auf der Standard-
                                ausgabe

...

sscanf(buffer,"Das ist ein Textsstring mit der Hexzahl  %x",&ausg);

                                die ASCII-Zeichen der Ziffernkette, die nach der
                                Zeichenfolge ...zahl in der variable buffer steht, wird
                                der Variable ausg zugewiesen
```

Hinweis: Die Argumente des sscanf-Befehls sind Zeiger auf Variablen

4. Funktionen

Funktionen dienen der Mehrfachausnutzung von Programmkode und führen zu einer Strukturierung und Erhöhung der Übersichtlichkeit des Quellprogramms. Sie haben den folgenden Aufbau:

Ergebnistyp Funktionsname (Parameterliste)
{
Deklarationen
Anweisungen
}

Beispiel:

```
float kreisberechnung(float radius);          Definition einer Funktion mit dem Namen
  {                                           kreisberechnung vom Typ float
 # define PI 3.1415
 return(PI* radius* radius);
 }
```

Innerhalb einer Funktion können Deklarationen vorgenommen werden, deren Gültigkeit sich nur auf das Unterprogramm bezieht. Ergebnistyp ist dabei ein bereits vordefinierter oder vom Programmierer selbst definierter Datentyp. Dem Funktionsnamen bzw. dem Namen/ Bezeichner der Funktion folgt eine Parameterliste, welche die von der Funktion übernommenen Variablen mit Variablentyp und Variablennamen beinhaltet. Der Aufruf der Funktion erfolgt durch die Zuweisung des Rückgabewertes der Funktion an eine Variable. Dazu wird der Name der Funktion mit der Parameterliste als Right-Value niedergeschrieben.

Beispiel:

```
float wert,a;
...
wert = keisberechnung(a);
```

Die Reihenfolge der Variablen in der Deklaration bzw. Definition der Funktion muss der Reihenfolge der Variablen später beim Aufruf entsprechen. Insbesondere müssen die Datentypen der Variablen, die an die Funktion übergeben werden, mit den Datentypen in der Funktionsvereinbarung übereinstimmen. Ebenso sollte der Datentyp des Rückgabewertes der Funktion dem Typ der Variable entsprechen, welcher der Rückgabewert zugewiesen wurde.

Unter Umständen werden keine Parameter an die Funktion übergeben. Dann ist die Parameterliste beim Funktionsaufruf leer. Wird hingegen der Rückgabewert verworfen, so steht der reservierte Bezeichner void als Ergebnistyp. Erfolgt kein Zuweisen des Rückgabewerte der Funktion, so ist diese automatisch vom Typ void.

Beispiel:

```
void pause(void);                 Definition einer Funktion pause
{....}

pause();                          Aufruf einer Funktion pause
```

Spätesten beim ersten Funktionsaufruf muss dem Compiler die Funktion bekannt sein. Deshalb ist es erforderlich, vor einem Funktionsaufruf dem Compiler die Funktionsdefinition oder mindestens die Funktionsdeklaration bekannt zu machen. Funktionen können sich nur dann selbst aufrufen bzw. von mehreren Stellen gleichzeitig genutzt werden, wenn sie reentrant sind. Reentrante Funktionen erfordern die Beachtung besonderer Randbedingungen bei ihrer Programmierung wie die Stacküberwachung, die Nutzung spezieller Übergabespeicherbereiche usw..