MODUL „C“
Programmierung in „C“

Version 1.1 Dipl.-Ing. Dr. Josef Humer 1996

BULME GRAZ

INHALTSVERZEICHNIS
MODUL „C“
Programmierung in „C“

Inhalt Seite

1 Einführung **4**
2 Wichtige Begriffe **5**
3 Programmierung in „C“ Allgemein **9**
3.1 Grundgerüst der Programmiersprache C 9
3.2 Aufbau eines C - Programmes 9
3.2.1 Preprozessor - Anweisung 10
3.2.2 Deklarationen und Definitionen 10
3.2.3 Funktionen mit lokalen Definitionen 10
3.3 Datentypen 11
3.3.1 Gültigkeitsbereich von Variablen 11
3.3.1.1 Globale Variablen 11
3.3.1.2 Formale Parameter 11
3.3.1.3 Lokale Variablen 11
3.3.2 Speicherklassen 12
3.3.2.1 Automatic 12
3.3.2.2 Static 12
3.3.2.3 Extern 12
3.3.2.4 Register 12
3.3.3 Datentypen 13
3.3.3.1 Einfache Datentypen 13
3.3.3.2 Zusammengesetzte Datentypen 14
3.3.3.3 Abgeleitete Datentypen 16
3.4 Ausgaben 20
3.4.1 Die Funktion printf() 20
3.4.2 Die Funktionen puts() und putchar() 22
3.5 Eingaben 22
3.5.1 Die Funktion scanf() 23
3.5.2 Die Funktionen gets() und getch() 24
3.6 Operationen und Operatoren 26
3.6.1 Definitionen und Übersicht 26
3.6.2 Zuordnungen 26
3.6.3 Unäre und Binäre Operatoren 27
3.6.4 Relationale Operatoren 27
3.6.5 Logische Operatoren 28
3.6.6 Bitmanipulatoren 28
3.6.7 Kombinatorische Operatoren 29
3.6.8 Adress-Operatoren 30
3.6.9 Komplexe Ausdrücke 30
3.7 Selektion 31
3.7.1 IF .. ELSE 31
3.7.2 SWITCH .. CASE 32

3.8 Iteration mit Schleifen 34
3.8.1 WHILE - Schleife 34
3.8.2 FOR - Schleife 35
3.8.3 DO .. WHILE Schleife 35
3.9 Zusätzliche Steuerung des Programmablaufes 36
3.9.1 Befehl Return 36
3.9.2 Befehl Break 36
3.9.3 Befehl Continue 37
3.10 Typkonvertierung 38
3.10.1 Automatische Konvertierung 38
3.10.2 signed ⇔ unsigned 38
3.10.3 char ⇒ int 39
3.10.4 int ⇒ float 39
3.10.5 int ⇒ float 40
3.10.6 long ⇒ float 40
3.10.7 float ⇒ int 40
3.10.8 float ⇒ long 41
3.10.9 cast - Operator 41
4 C51-Datentypen 42
4.1 Globale Variablen 43
4.2 Lokale Variablen 43
4.3 Speicherklassen 44
4.3.1 auto 44
4.3.2 static 44
4.3.3 extern 45
4.3.4 register 45
4.4 Speichertypen 46
5 Handhabung des C51 - Compilers 48
5.1 Aufruf 48
5.2 Steurerparameter: 48
5.2.1 SRC 49
5.2.2 Optimize 50
5.2.3 Define 50
5.2.4 Symbols (SB) 50
5.2.5 Code 50
5.2.6 DEBUG 50
5.2.7 Speichermodelle 50
5.2.7.1 small 50
5.2.7.2 compact 51
5.2.7.3 large 51
5.2.8 INTVECTOR (IV) 51

1 Einführung

Ein **Compiler** ist ein Programm zur Übersetzung eines in einer höheren Programmiersprache geschriebenen Programmes in die Maschinensprache. Dieses Programm ist jedoch noch nicht lauffähig. Damit das Programm ausgeführt werden kann, muß ein Linker verwendet werden.

Beispiel:
c:\work>**C51 <filename>.c51**
Erklärung:

C51	Programmaufruf
<filename>.c51	Quelldatei in der Programmsprache „C“

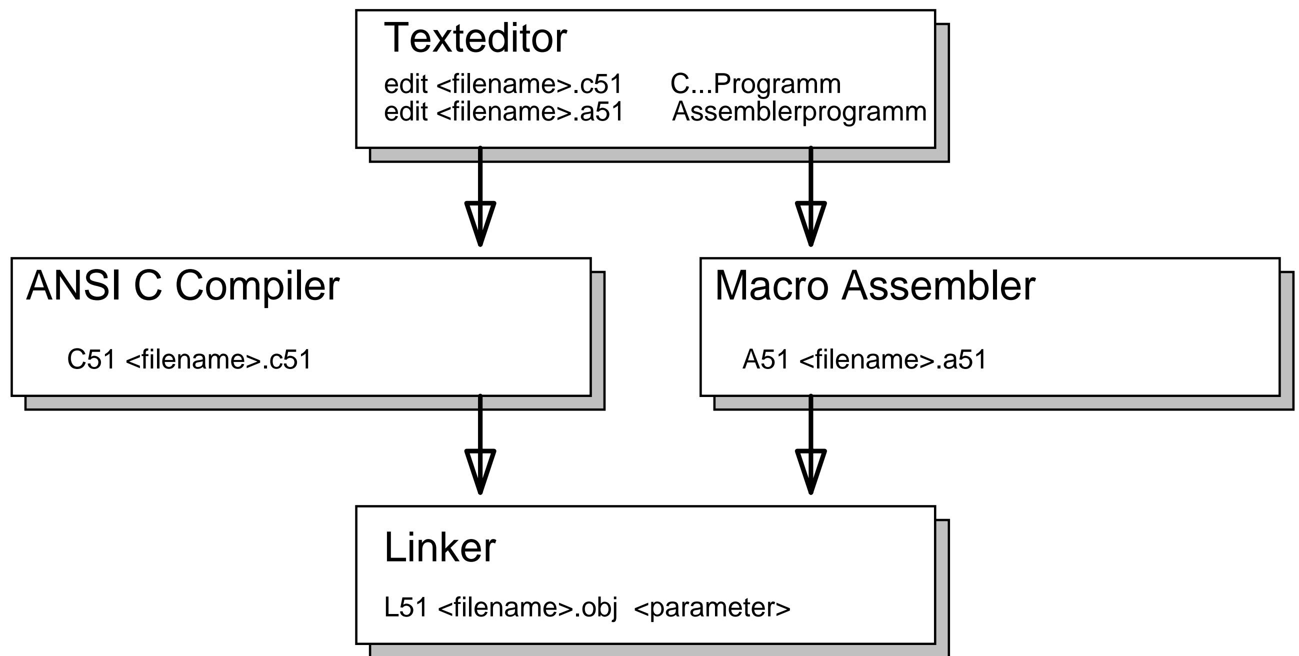
Der **Linker** fügt Programmteile aneinander, wandelt die relativ auf das Modul bezogenen Adressen in absolute Adressen und bindet Routinen aus den Bibliotheken dazu.

Beispiel:
c:\work>**L51 <filename>.obj code(8000H) xdata(0C000H) RAMSIZE(256)**
Erklärung:

L51	Programmaufruf Linker - Keil
<filename>.obj	Objektdatei
RAMSIZE(256)	Interner Speicher 256 BYTE
code(8000h)	Programm soll ab 8000HEX gespeichert werden
xdata(0C000H)	Externer Arbeitsspeicher, beginnend mit ..

Von einem **Cross-Compiler** spricht man, wenn der Code, der vom Compiler erzeugt wird, nicht auf dem System ablaufen kann, auf dem er erstellt wurde.

Blockschaltbild für Editor, Compiler und Linker



2 Wichtige Begriffe

Adresse:

Zahl oder symbolischer Name zur Identifizierung eines Registers, Speicherwortes oder Speicherbereiches, Eingabe- und Ausgabe-Kanals.

ANSI-C:

Eine Definitionsnorm für die Programmiersprache C. ANSI-C wurde 1988 von der American National Standards of Institute definiert.

Anweisung:

Eine Anweisung kann eine Operation oder ein Funktionsaufruf sein. Jede Anweisung wird mit einem Semicolon abgeschlossen.

Argument:

Werden bei einem Funktionsaufruf Werte übergeben, so wird von einer Argumentübergabe gesprochen.

Assembler:

Ein Programm zur Übersetzung eines in Assemblersprache geschriebenen Programms in ein auf dem Zielprozessor ablauffähiges Maschinenprogramm.

Compiler:

Programm zum Übersetzen von Programmen aus einer höheren Programmiersprache in einen Maschinencode.

Cross Compiler:

Ein Compiler, der das Programm in eine Maschinensprache eines anderen Zielprozessors übersetzt. In unserem Falle läuft ein C51-Compiler oder C166-Compiler auf den IBM- oder IBM-kompatiblen Personal-Computer.

Definition:

Bei der Definition von Variablen wird dem Compiler der Datentyp bzw. Speichertyp bekanntgegeben und Speicherplatz dafür reserviert.

Deklaration:

Einem Programmodul wird eine Variable, Funktion usw. bekanntgegben. Dies muß erfolgen, wenn z.B. Funktionen verwendet werden, die nicht im aktuellen Modul stehen.

Funktion:

Ein Programm in C besteht aus einer oder mehreren Funktionen, die in einem oder in mehreren Quelltextmodulen untergebracht sein können. Alle zu einer Funktion gehörenden Anweisungen müssen in geschweifte Klammern eingeschlossen werden.

Beispiel einer Funktion:

```
main()                  /* Funktionsname.  Die Klammer hinter dem   */
                        /* Funktionsnamen dient der Parameterübergabe*/

{                       /* Beginn der Funktion */
   char a;              /* Definition einer Variablen */
                        /* Eine Sourcezeile in C wird mit einem */
                        /* Semikolon abgeschlossen */
   a = 10;              /* Wertzuweisung */
}                       /* Ende der Funktion */
```

Linker:

Ein Programm des z.B. C51-Programmpaketes wird zum Binden mehrerer zusammengehörender, getrennt übersetzter Programme zu einem ablauffähigen Programm verwendet.

main:

In jedem C-Programm muß mindestens eine Funktion mit dem Namen main vorhanden sein, da hier die Ausführung beginnt.

Modul:

Ein File, das dem Compiler zum Übersetzen angegeben werden kann. Ein Programm kann in mehrere Programmodule aufgeteilt werden, um die Übersichtlichkeiten zu erhöhen.

Aufbau eines Moduls:

```
/* Modulname: XXXXXXXX.C51    Ausdruck am: 14.05.1992      */
/* Funktion :   division      Version 1.1                  */
/* User     :   NONAME                                     */
/* Beschreibung:                                           */
/*     Dividiert zwei Variablen.                           */
/* Änderungen:                                             */
/*     1.1  Überprüfung, ob durch 0 dividiert wird         */
/* ******************************************************  */

#pragma symbols            /* Globale Steueranweisungen    */

#include <reg515.h>        /* Laden von Deklarationen      */

int a,b,erg;               /* Definition von Variablen     */

main() {                   /* Funktionsdefinition          */

  a = 3;                   /* Wertzuweisungen              */
  b = 4;

  if (b != 0)              /* Abfrage, ob der Divisor ungleich 0 ist   */
      erg = a / b;         /* dann wird die Berechnung durchgeführt.   */

 }
```

Operation:

Arithmetischer oder logischer Ausdruck.

Parameter:

Wird bei einem Funktionsaufruf ein Wert mit übergeben, so ist dieser Wert ein Parameter. Ein Parameter kann z.B. eine Konstante, eine Variable, eine Struktur, ein Pointer usw. sein.

Simulator:

Ein Simulator ist ein Programm, das einen Zielprozessor mittels Software nachbildet. Es können damit Programme oder Zielhardware getestet werden.

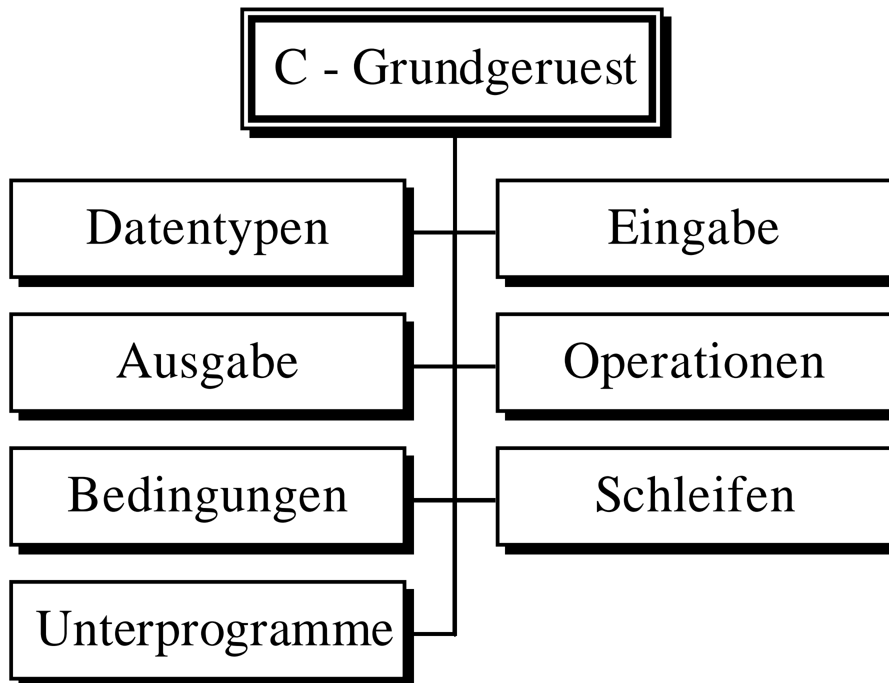
Emulator:

Die Software wird direkt auf der Zielhardware getestet.

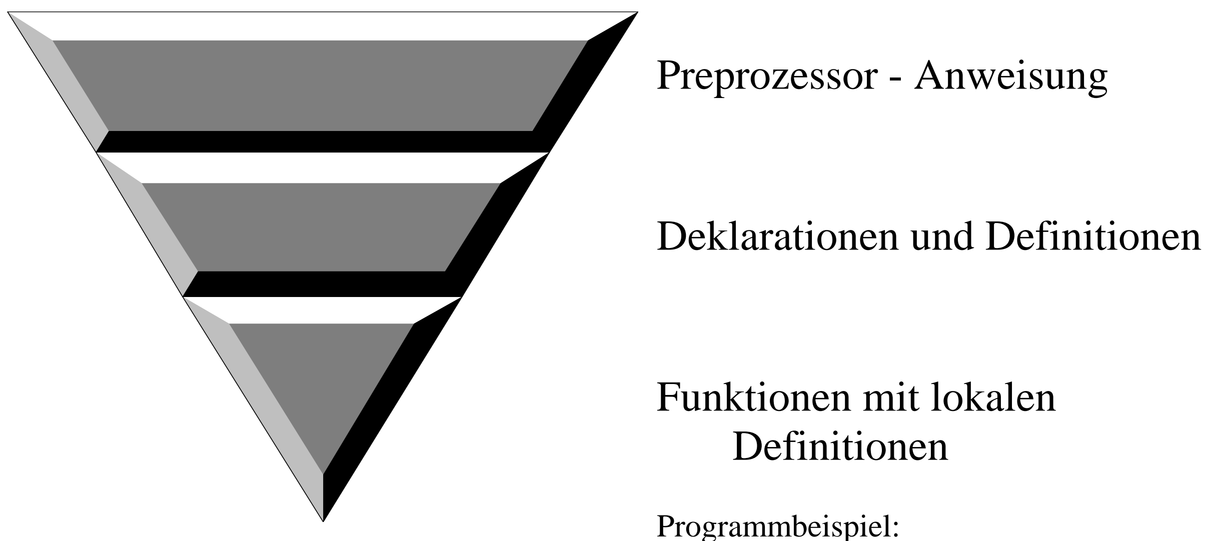
3 Programmierung in „C“ Allgemein

Im folgenden wird die Programmiersprache „C“ ganz allgemein betrachtet. Die folgenden Beispiele sind auf dem PC unter Turbo C oder Borland C lauffähig. Auf Änderungen gegenüber dem Keil C - Compiler wird später eingegangen.

3.1 Grundgerüst der Programmiersprache C



3.2 Aufbau eines C - Programmes



Programmbeispiel:

```
/* Modulname: first.C51    Ausdruck am: 14.05.1992         */
/* Funktion :   ausgabe       Version 1.1                   */
/* User     :   NONAME                                      */
/* Beschreibung:                                            */
/*     Gibt eine Zeichenfolge aus                           */
/* Änderungen:                                              */
/*/* ******************************************************      */

#include <stdio.h>                /* Preprozessoranweisung */

void main()                       /* Funktion - Hauptprogramm */
{
    puts(„Guten Tag, C-Freak!“);   /* Zeichenfolge ausgeben */
}
```

3.2.1 Preprozessor - Anweisung

Der Compiler enthält einen Preprozessor, einen unkomplizierten, aber wirkungsvollen Vorübersetzer, welcher unabhängig vom Compiler arbeitet.

Beispiel: *# define <name>*
 # include <dateiname>

Die Aufgaben des Preprozessors sind:

- Einbinden anderer Quelldateien
  - Bedingte Übersetzung von Programmteilen
    - Ersetzung von Namen
      - Parameterabhängige Ersetzung von Namen

3.2.2 Deklarationen und Definitionen

Die Deklaration macht etwas bekannt. Sie reserviert keinen Speicherplatz, sie erfüllt nur die Aufgabe, einen Namen bekannt zu machen.

Die Definition legt ein Objekt physikalisch an. Es wird Speicherplatz reserviert und das Objekt eventuell auch initialisiert.

3.2.3 Funktionen mit lokalen Definitionen

Unter einer Funktion ist eine Sammlung von Befehlen und Anweisungen (meist auch wieder Funktionen) zu verstehen, die mit einem beliebigen Funktionsnamen angesprochen werden.

Kennzeichen einer Funktion:

Der Name und die direkt folgende öffnende und schließende runde Klammer.

```
int summe()
{ .... }
```

3.3 Datentypen

Variablen müssen grundsätzlich vor ihrer erstmaligen Verwendung definiert werden. Dabei sind 3 Punkte von Bedeutung:

- An welcher Stelle des Programmes die Definition erfolgt. Damit wird der Gültigkeitsbereich einer Variablen bestimmt.
- Die Angabe einer Speicherklasse. Damit werden die dynamischen Eigenschaften (Lebensdauer etc.) festgelegt.
- Die Zuordnung zu einem Datentyp. Dies ermöglicht dem Compiler, den Speicherbedarf zu berechnen und Speicherplatz zu reservieren.

3.3.1 Gültigkeitsbereich von Variablen

3.3.1.1 Globale Variablen

Gültigkeitsbereich: Gesamtes Programm
Definition: außerhalb von Funktionen

Beispiel:

```
int a,b;

main()
{
     Anweisungen;
}
```

3.3.1.2 Formale Parameter

Gültigkeitsbereich: innerhalb der Funktion
Definition: Nach dem Funktionsnamen vor Beginn der Funktion
Beispiel:

```
int summe(a,b)    /* Funktion summe gibt einen int-Wert zurück */
int a,b;
{
     Anweisungen;
}
```

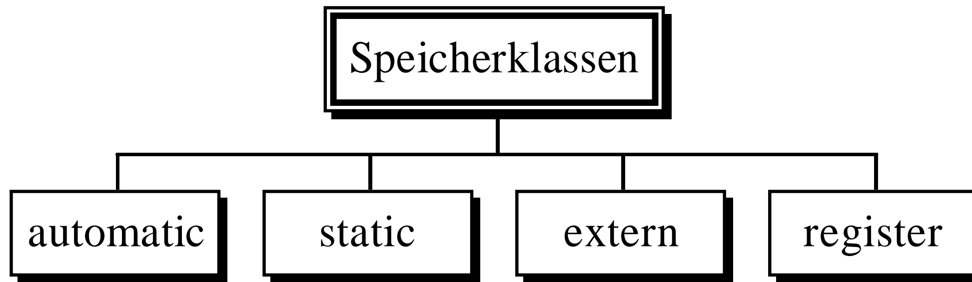
3.3.1.3 Lokale Variablen

Gültigkeitsbereich: innerhalb der Funktion
Definition: innerhalb der geschweiften Klammern einer Funktion

Beispiel:

```
int summe()
{
int a,b;
     Anweisungen;
}
```

3.3.2 Speicherklassen



In C gibt es für Variable **vier Speicherklassen**, die Angabe ist optional. Wird bei der Definition einer Variablen keine Speicherklasse angegeben, wird vom Compiler die Standardvorgabe eingesetzt.

3.3.2.1 Automatic

Ist die **Standardvorgabe für lokale** Variablen. Sie werden beim Start nicht initialisiert und verlieren ihren Wert, wenn die Funktion verlassen wird.

3.3.2.2 Static

Ist die **Standardvorgabe für globale** Variablen. Sie werden beim Start des Programmes auf 0 gesetzt, wenn ihnen nicht explizit ein Wert zugeordnet wird. Auch lokale Variablen können als static definiert werden, sie werden dann ebenfalls mit 0 initialisiert und behalten ihren Wert zwischen zwei Aufrufen ihrer Funktion.

Beispiel:

```
int summe()
{
static int a,b;
     Anweisungen;
}
```

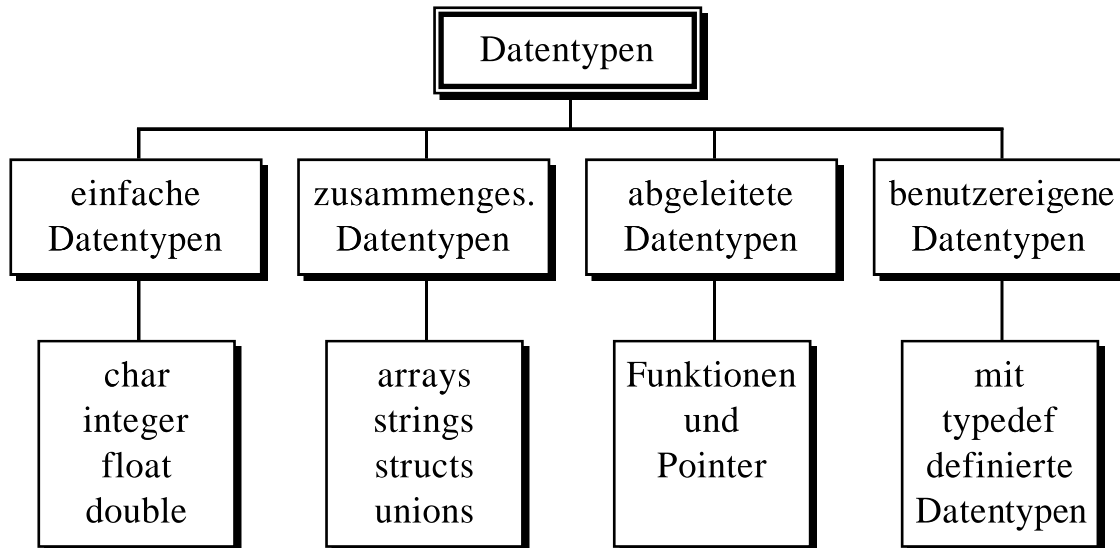
3.3.2.3 Extern

Ist nur erforderlich, wenn die Variable auch in anderen, getrennt compilierten Modulen verwendet wird. Dies ist nur bei globen Variablen möglich.

3.3.2.4 Register

Wird vom Compiler automatisch für jene lokalen Variablen verwendet, auf die rasch zugegriffen werden soll.
(für PC werden die CPU-Register DI und SI verwendet,
für 8051 werden die internen Register verwendet.)

3.3.3 Datentypen



3.3.3.1 Einfache Datentypen

⌦ **char**

Datenlänge: 8 bit
umfaßt ein einzelnes Zeichen und benötigt ein Byte Speicherplatz
Wertebereich: -128 ... 0 ... +127
unsigned char: 0 ... 255 (2^8)

⌦ **integer**

Datenlänge: 2 Byte
zur Speicherung ganzer Zahlen,
Wertebereich: -32786 ... 0 ... 32767
unsigned int: 0 ... 65536 (2^16)

⌦ **long**

Datenlänge: 4 Byte
Ist ein Integertyp mit 4 Byte Länge
Wertebereich: -2147483648 ... 0 ... 2147483647
unsigned long: 0 ... 4294967295 (2^32)

⌦ **float**

Datenlänge: 4 Byte
für rationale Zahlen mit einfacher Genauigkeit. 7 signifikante Ziffern !
Wertebereich: 3,4 E-38 ... 0 ... 3,4 E+38

⌦ **double**

Datenlänge: 8 Byte
für rationale Zahlen mit erhöhter Genauigkeit. 16 signifikante Ziffern !
Wertebereich: 1,7 E-308 ... 0 ... 1,7 E+308

⌦ **long double (PC only)**

Datenlänge: 10 Byte
für höchste Genauigkeit (und optimale Nutzung des Coprozessors)
für rationale Zahlen mit erhöhter Genauigkeit.
Wertebereich: 3,4 E-4932 ... 0 ... 3,4 E+4932

3.3.3.2 Zusammengesetzte Datentypen

Diese Datentypen werden aus einfachen Datentypen zusammengesetzt. Der dabei entstehende Vorteil ist in der Ansprechbarkeit von einer ganzen Gruppe von Daten zu suchen.

⌦ **Arrays**

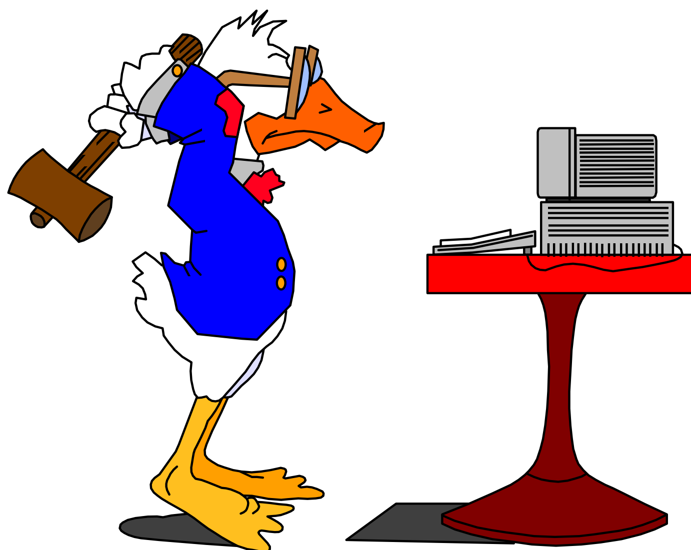
Ein Array stellt ein Feld gleicher Datentypen dar. Da die Sprachdefinition von C keine Strings als eigenen Datentyp kennt, werden hierfür Zeichenarrays verwendet.

Beispiel:

```
/* Variablendefinitionen */
char string[60];
....
```

Hinweis:

Bei einem Array mit 60 Elementen reicht der Index von 0 bis 59, nicht bis 60! Das erste Element wird mit string[0] angesprochen.



⌧ Strukturen

fassen mehrere Variablen unterschiedlichen Typs unter einem gemeinsamen Oberbegriff zusammen.

Beispiel:

```
/* Variablendefinition */
struct
{
     int KuNr;
     char name[30];
     double vermoegen;
} kunde;          /* Erweiterung auf ein Array kunde[XX] möglich */

/* Anwendung im Hauptprogramm */
kunde.KuNr = 367;
strcpy(kunde.name,"BULME");
kunde.vermoegen = 2156786399.16;
```

Achtung:

C unterscheidet zwischen Groß- und Kleinschreibung!

⌧ Verbunde

Ein Verbund (engl. union, auch als Variante bezeichnet) ist eine Variable oder eine Struktur, die mehrere Objekte verschiedener Datentypen und Größen enthält, aber nur für das größte Objekt Speicherplatz reserviert. Die verschiedenen Objekte überlagern einander.

Beispiel:

```
/* Für einen 8-bit Rechner soll ein 16 bit ADU-Wert verarbeitet werden.
     Im Register ADUL befinden sich bit0...bit7 und im Register ADUH
     befinden sich bit8 ... bit15. */

int ADWert;

main()
{
.....
     ADWert = ADUH*256+ADUL;
.....
}
```

Realisierung mit union:

```
/* Für einen 8-bit Rechner soll ein 10 bit ADU-Wert verarbeitet werden.
     Im Register ADUL befinden sich bit0...bit7 und im Register ADUH
     befinden sich bit8 und bit9. */

union first
{
     unsigned int gesamt;
     unsigned char wert[2];
}ADWert;


main()
{
.....
.....ADWert.wert[1] = ADUL;
     ADWert.wert[0] = ADUH;
/* Der Gesamtwert seht nun in der Variable ADWert.gesamt zur Verfügung */
}
```

Darstellung der Speicherstellen:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	°0
LSB = ADWert.wert[1] Speicher (x+1)								ADWert.wert[0] Speicher (x) =MSB							
ADWert.gesamt															

3.3.3.3 Abgeleitete Datentypen

⌧ **Funktionen**

In C wird auch eine Funktion einem Datentyp zugeordnet. Er entspricht dem Wert, der von der Funktion zurückgeliefert wird.

Beispiel:

```
int summe();       /* Rückgabewert ist vom Typ Integer */
void lcd();        /* kein Rückgabewert */
```

⌧ **Pointer**

Pointer sind Zeiger auf Speicherstellen und werden in C sehr viel und gerne verwendet. Sie können die Adressen von Variablen beliebigen Typs, die Anfangadressen von Funktionen oder auch die Speicheradressen weiterer Zeiger enthalten. Man kann mit ihnen aber auch auf einen beliebigen Speicherbereich (Bildschirmspeicher am PC) zugreifen.

Je nach Speicherbereich sind Pointer near (2 Bytes) oder far bzw. huge (4 Bytes). Wird der Typ nicht extra angegeben, gilt die Compilereinstellung.

Bei dem C-51-Compiler sind zwei verschiedene Arten von Pointern möglich.

- ⌧ **Generic - Pointer**
- ⌧ **memory specific Pointer**

Generic - Pointer:

Mit dem generic pointer lassen sich alle Speicherbereiche (mit Ausnahme des bdata-Bereichs) ansprechen. Der jeweilige Speicherbereich des 8051, der durch den Pointer angesprochen werden soll, wird durch den Speichertyp festgelegt. Bei der Adreßzuweisung einer Variablen wird der Speichertyp automatisch eingetragen. Wird eine Konstante als Pointeradresse verwendet, so ist der Speichertyp expliziz mit anzugeben.

Speichertyp bei einer Konstante als Pointeradresse:

Code	Speicherbereich
01	idata
02	xdata
03	pdata
04	data
05	code

Beispiel: **Pointer-Skalierung**:

```
int *ptr_i;
char *ptr_c;

int erg, array[4]={1,2,3,4};

main()
{
     ptr_i = array;      /* Adresse des ersten Elements laden */
     ptr_c = array       /* Adresse des ersten Elements laden */

     erg = *ptr_i;       /* Der Wert 1 wird erg zugewiesen */
     erg = *ptr_c;       /* Da der Pointer vom Typ char ist, liefert
                              er den MSB-Anteil vom ersten Element */
                         /* in diesem Fall 0 */

     ptr_i++;            /* Der Pointer zeigt nun auf das 2 Element
                              im Array */
     ptr_c++;            /* siehe Bild */

     erg = *ptr_i;       /* Der Wert 2 wird erg zugewiesen */
     erg = *ptr_c;       /* Der Wert 1 wird erg zugewiesen, da der
                              ptr_c nur um ein Byte erhöht wurde */

}
```

Adresse im RAM	Inhalt	Pointer
x+3	array[1] LSB	
x+2	array[1] MSB	⌫ ptr_i (neu)
x+1	array[0] LSB	
x	array[0] MSB	⌫ ptr_i (alt)

Adresse im RAM	Inhalt	Pointer
x+3	array[1] LSB	
x+2	array[1] MSB	
x+1	array[0] LSB	⌫ ptr_c (neu)
x	array[0] MSB	⌫ ptr_c (alt)

Beispiel: **Verwendung einer Konstanten als Pointer**

```
#define Port_B ((unsigned char *) 0x28001L)
      /* Adresse von Port_B = 8001 hex im externen RAM, siehe Tabelle */

main()
{
     *Port_B = 0x80;
      /* An die Adresse 0x8001 im externen Speicherbereich wird der Wert 80
     hex
            geschrieben */
}
```

Funktionsaufruf über pointer!

Mit Pointern lassen sich Funktionen indirekt aufrufen. Es wird die Einsprungadresse einer Funktion an einen Pointer übergeben. Somit besteht die Möglichkeit, ein Unterprogramm als Paramter zu übergeben.

Beispiel:

```
int timer1(void)
{
     return(TH0*256+TL0);
}

int (*ptr)();

main()
{
     int wert;
     ptr = &timer1;      /* Einsprungadresse der Funktion */
     wert = (*ptr)();    /* Funktionsaufruf */
}
```

Memoric specific pointer

Der Unterschied zwischen dem memory specific pointer und dem generic pointer besteht darin, daß der Speichertyp, auf den der Pointer zeigt, beim memory specific pointer schon im Sourcecode bekannt ist. Der Vorteil liegt im geringeren Code-Aufwand.

Beispiel:

```
#pragma small code sysmbols debug ot(1)
ptr_test()
{
     data char data *ptr;
     data char a;
     ptr = &a;
}

oder;

     xdata char data *ptr;
     data char a;
     ptr = &a;
```

Pointervariable zur Adreßspeicherung:

Soll eine Speicherzelle mittels eines Pointers angesprochen werden, so ist der Pointer zuerst mit der Adresse der Speicherstelle zu laden. Der &-Operator liefert die Adresse einer Variablen bzw. einer Funktion. Dabei ist zu beachten, daß der Datentyp vom Pointer mit dem der Variablen übereinstimmen muß. Stimmen sie nicht überein, so liefert der C51-Compiler eine Warnung.

Indirektes Beschreiben des Speichers mit einem Pointer:

*pointer = Variablenname;

Indirektes Lesen des Speichers mit einem Pointer:

Variablenname = *pointer;

Pointer - Arithmetik:

einige Beispiele:

erg = (*pointer)++;

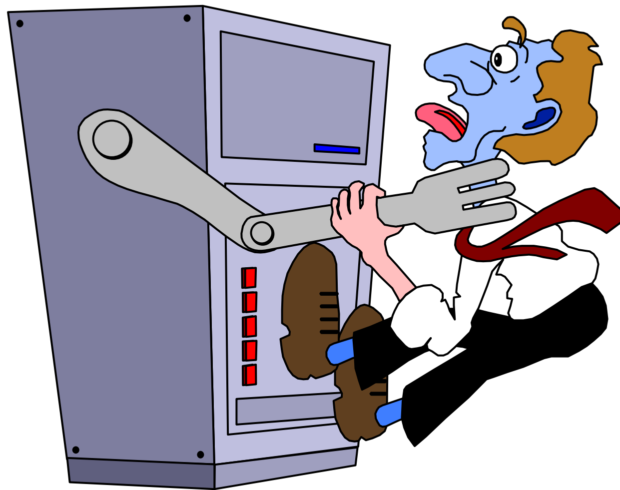
Zuerst wird der Inhalt der Adresse erg zugewiesen und danach wird nicht der Pointer erhöht, sondern der Inhalt auf den der Pointer zeigt;

erg =++(*pointer);

Zuerst wird der Inhalt, auf den pointer zeigt, erhöht und danach wird der Inhalt erg zugewiesen.

erg = *pointer++;

Zuerst wird der Inhalt der Adresse erg zugewiesen und danach wird pointer inkrementiert.



3.4 Ausgaben

Als Ausgaben wird alles bezeichnet, was ein Programm an Daten von sich gibt, unabhängig davon, ob es sich dabei um Text oder Grafik handelt, die zu Bildschirm oder Drucker geschickt oder in eine Datei auf Diskette oder Festplatte geschrieben werden.

In diesem Skriptum wird nur die Ausgabe auf dem Bildschirm bzw. für 8051 gleichwertig mit der seriellen Schnittstelle behandelt.

3.4.1 Die Funktion printf()

Am häufigsten wird in C die Ausgaberoutine printf() verwendet, weil diese Funktion nicht nur Zeichen auf den Bildschirm schreiben kann, sondern auch in der Lage ist, Daten umzuwandeln und formatiert auszugeben.

Syntax:

printf(„Format-String“, <Wert1, ..., Wertn>);

⌧ **Format-String**

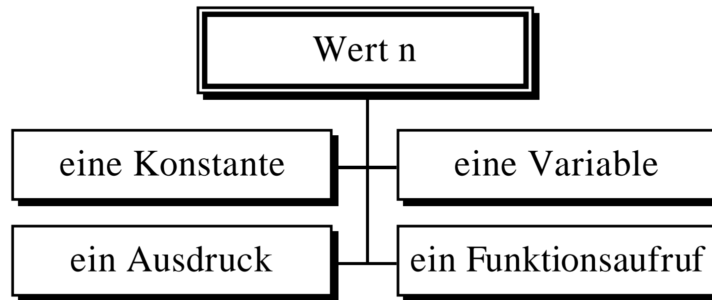
Der Format-String ist eine Folge von Zeichen, die in Anführungszeichen eingeschlossen werden müssen. Die Funktion printf() schreibt alle Zeichen genauso am Bildschirm mit Ausnahme der Zeichen **/** und **%** und darauffolgende Zeichen, die eine besondere Bedeutung haben.

Beispiel:

```
printf(„Hallo hier bin ich!“);
                  /* Ausgabe am Bildschirm: Hallo hier bin ich! */
                  /* für 8051 Ausgabe an der seriellen Schnittstelle */

printf(„Das Ergebnis ist: %d \n“, 47);
            /* Ausgabe am Bildschirm:  Das Ergebnis ist: 47 */
            /* %d ...   Ausgabe einer Integerzahl           */
            /* \n ...   Neue Zeile nach Ausgabe             */
```

⌧ **Wert**

Als Wert kann stehen:



Beispiel:

```
printf(„Die Summe von %d und %d ist %d\n", x, y, x+y);
     /* Die Variablen x und y müssen vom Typ Integer sein */
```

⌧ Formatelemente

%d	decimal	Integerwert (16 bit) mit Vorzeichen
%i	integer	Integerwert (16 bit) mit Vorzeichen
%ld	long decimal	Integerwert (32 bit) mit Vorzeichen
%u	unsigned	vorzeichenloser Integerwert
%x u. %X	hex	Integer in hexadezimaler Schreibweise
%O	octall	Integer in oktaler Schreibweise
%p	pointer	Zeiger bzw. Speicheradresse
%f	float	Gleitkommadarstellung mit Vorzeichen
%e	exponential	Gleitkomma in wissenschaftlicher Notation (m.n E±xx)
%c	character	ein einzelnes Zeichen
%s	string	eine Zeichenkette

Beispiel:

```
printf(„%4d",17);
     /* Gibt den Wert 17 rechtsbündig in einem 4 Zeichen breiten Feld aus.
            2 Leerzeichen werden automatisch vorangestellt.
            Ist das Feld zu klein, wird die Angabe 4 ignoriert */

printf(„%7.2f",67.189);
     /* Gibt den Wert 67.189 in einem Feld von 7 Zeichen Breite aus.
     Es werden 2 Leerzeichen vorangestellt, die Anzahl der Nachkomma wird
     auf 2 fixiert. Achtung der Dezimalpunkt zählt als eine Stelle. */

printf(„Steigerungsrate %5.2f%%",14.3);
     /* Ergibt als Ausgabe: Steigerungsrate 14.30%
            für die Anzeige von % muß ein zweites %-Zeichen vorangestellt
            werden. */
```

⌧ Steuerzeichen

Steuerzeichen werden mit dem Backslash eingeleitet.

Zusammenstellung:

\b	backspace	Cursor ein Zeichen nach links
\f	formfeed	Seitenvorschub für Drucker, oder Bildschirm löschen
\n	newline	Zeilenvorschub
\t	tab	Tabulator
\xhh	hex	stellt ein Zeichen dar, wobei hh für den ASCII-Code steht (Angabe in HEX)
\DDD		wie vorhin, aber in 3 Oktalziffern
\\		stellt den Backslash selbst dar
%%		gibt das %-Zeichen aus

3.4.2 Die Funktionen puts() und putchar()

Diese beiden Sonderfälle sind häufig verwendete Sonderfälle von printf(). Sie sind nicht so leistungsfähig wie ihr Vorbild, dafür aber schneller und beanspruchen weniger Speicherplatz.

Beispiel:

```
/* Variablendefinition */
char zeichen;
char name[30];

/* Hauptprogramm */
main()
{
     .....
     zeichen = „A";
     name=„Humer";
     putchar(zeichen);  /* entspricht dem printf(„%c",zeichen); */
     puts(name);        /* entspricht dem printf(„%s\n",name);
                        Zeilenvorschub wird automatisch angehängt */
}
```

3.5 Eingaben

In C sind verschiedene Eingabefunktionen vorhanden, mit deren Hilfe Daten von der Tastatur, aus einer Datei von Diskette/Festplatte oder von der seriellen Schnittstelle eingelesen werden können.

In diesem Modul werden nur Eingaben von der Tastatur bzw. bei 8051 über die serielle Schnittstelle behandelt.

3.5.1 Die Funktion scanf()

Diese Funktion ist das Gegenstück zu printf() und wird ähnlich universell verwendet, weil sie nicht nur Zeichen von der Tastatur (RS232 bei 8051) einlesen kann, sondern auch gleich eine Konvertierung in das Speicherformat von C vornimmt.

Syntax:

scanf(„Format-String“,<Adresse1>,<Adresse2>,....);

⌦ **Format-String**

Der Format-String ist wie bei printf() eine Folge von Zeichen, die in Anführungszeichen eingeschlossen werden müssen. Es dürfen bei scanf() jedoch nur Format-Elemente und Whitespace enthalten sein.

⌦ **Formatelemente**

%d	decimal	Zeiger auf Integerwert (16 bit) mit Vorzeichen
%D	long decimal	Zeiger Integerwert (32 bit) mit Vorzeichen
%G	double	Zeiger auf doppelt genaue Fließkommazahl
%u	unsigned	Zeiger auf vorzeichenlosen Integerwert
%x u. %X	hex	Zeiger auf Integer in hexadezimaler Schreibweise
%p	pointer	Zeiger bzw. Speicheradresse
%f	float	(Zeiger) Gleitkommadarstellung mit Vorzeichen
%e	exponential	(Zeiger) Gleitkomma in wissenschaftlicher Notation (m.n E±xx)
%c	character	Zeiger auf ein einzelnes Zeichen oder char - Array, wenn die Breite angegeben wurde
%s	string	Zeiger auf eine Zeichenkette

Beispiel:

```
/* Modulname: XXXXXXXX.C51    Ausdruck am: 14.05.1994      */
/* Funktion :   XXXX        Version 1.1                    */
/* User     :   NONAME                                     */
/* Beschreibung:                                           */
/*     Addiert zwei Zahlen.                                */
/* Änderungen:                                             */
/* ******************************************************  */

#include <stdio.h>
/* Variablendefinition */
float zahl1,zahl2;

/* Hauptprogramm */
void main()
{
     printf(„Geben Sie 2 Zahlen ein:\n“);
     scanf(„%f %f“,&zahl1,&zahl2);
     printf(„Die Summe der Zahlen = %f\n“,zahl1+zahl2);
}
```

3.5.2 Die Funktionen gets() und getch()

Diese beiden Funktionen sind nicht nur Spezialfälle von scanf(), sondern gleichen auch einige Schwächen ihres Vorbildes aus. Sie sind zwar nicht so leistungsfähig, da sie keine Daten konvertieren können, dafür aber schneller und kompakter.

Programmbeispiel:
Versuchen Sie bei dem folgenden Programmbeispiel, einmal Vorname und Nachname, einmal nur Nachnamen einzugeben. Wie verhält sich dieses Programm?

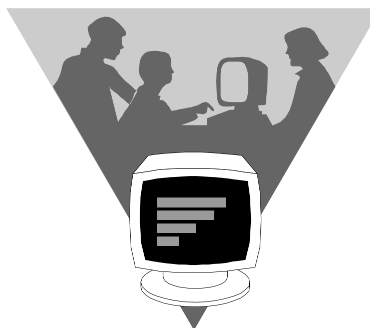
```
#include <stdio.h>
void main()
{
     char name[60];
     printf(„\nWie heißen Sie?  “);
     scanf(„%s“,name);
     printf(„Guten Tag, Herr/Frau %s\n“,name);
}
```

Die Schwäche des obigen Programms kann mit Hilfe der Funktion gets() eliminiert werden.

```
#include <stdio.h>
void main()
{
     char name[60];
     printf(„\nWie heißen Sie?  “);
     gets(name);
     printf(„Guten Tag, Herr/Frau %s\n“,name);
}
```

⌦ **gets()**
liest solange ein Zeichen ein, bis die Eingabe mit Return beendet wird. Die Zeichen werden an der angegebenen Adresse gespeichert, an das Ende wird ein ASCII-Null (\0) angehängt. Auch ein leerer String kann übernommen werden, wenn nur Return gedrückt wird. Der Zeilenvorschub wird nicht mitgespeichert.

⌦ **getch()**
liest ein einzelnes Zeichen von der Tastatur, ein Return ist nicht notwendig. Das Zeichen wird nicht von getch() auf den Bildschirm ausgegeben.



Programmbeispiel:

```
#include <stdio.h>
#include <conio.h>                  /* für PC wegen getch() und putch() */

void main()
{
     int ch;
     int i = 0;
     printf(„Geben Sie drei Zeichen ein: „);
     while (i++ < 3)                /* Schleife für 3 fachen Durchlauf */
     {
            ch = getch();           /* liest ein Zeichen */
            ch++;                   /* Zeichen wird um 1 erhöht (ASCII) */
            putch(ch);              /* Schreibe Zeichen auf Bildschirm */
     }
     getch();                       /* Wartet auf Tastendruck */
}
```

3.6 Operationen und Operatoren

Operationen sind Prozesse, wo bestimmte Symbole einen oder mehrere Werte miteinander verknüpfen. Die Symbole nennt man Operatoren.

3.6.1 Definitionen und Übersicht

Gruppe	Operator	Bedeutung
Zuordnungen	=	Zuweisung
Arithmetische Operatoren	*	Multiplikation
	/	Division
	%	Modulo
	+	Addition
	-	Subtraktion
	++	Inkrement
	--	Dekrement
Rationale Operatoren	>	groesser
	>=	groesser gleich
	<	kleiner
	<=	kleiner gleich
	==	gleich
	!=	ungleich
Logische Operatoren	&&	logisches UND
	||	logisches ODER
	!	logische Negation
Bitmanipulatoren	<<	Linksschieben
	>>	Rechtsschieben
	&	bitweises AND
	|	bitweises OR
	^	bitweises XOR
	~	bitweises NOT
Kombinierte Operatoren *(Operation und Zuweisung)*	+=	Addition
	-=	Subtraktion
	*=	Multiplikation
	/=	Division
	%=	Modulo
	>>=	Shift Right
	<<=	Shift Left
	&=	AND
	|=	OR
	^=	XOR

3.6.2 Zuordnungen

Zuordnungen stellen die grundlegendste Operation jeder Programmiersprache dar.

Beispiel:

```
/* Variablendefinitionen */
int a;
float b, summe;

a = 4;
b = 34.5;
summe = a + b;
a = b = summe;
                        /* Der Wert von summe wird b und a zugewiesen */
```

3.6.3 Unäre und Binäre Operatoren

Neben dem üblichen Satz binärer Operatoren, also

*	Multiplikation
/	Division
%	Modulo (Rest einer Division)
+	Addition
-	Subtraktion

unterstützt C ein unäres Minus (a+(-b)) zur Bildung des Zweierkomplements eines Operanden, und ein unäres Plus (a+(+b)).

Die Operatoren „Increment“ (++) und „Decrement“ (--) werden unär angewendet, das heißt, sie beziehen sich jeweils auf einen Operanden, der um eins erhöht bzw. erniedrigt wird. Es gibt zwei Möglichkeiten, vor und nach dem Operanden:

sum = a+ ++b;

Der Wert von b wird vor der Addition um 1 erhöht.

sum = a+ b++;

Der Befehl führt zuerst die Addition und Zuweisung aus und erhöht danach den Wert von b um eins.

Programmbeispiel:

```
#include   <stdio.h>

void   main()
{
         int a =  5,  b  5;
         int summe;
         char *format;
         format = "a  =%d b = %d  Summe = %d \n";
         summe  = a   + b;    printf(format,a,b,summe);
         summe  = a++ + b;    printf(format,a,b,summe);
         summe  = ++a + b;    printf(format,a,b,summe);
         summe  = --a + b;    printf(format,a,b,summe);
         summe  = a-- + b;    printf(format,a,b,summe);
         summe  = a   + b;    printf(format,a,b,summe);
}
```

3.6.4 Relationale Operatoren

Relationale Operatoren vergleichen zwei Operanden miteinander. Das zurückgelieferte Ergebnis ist TRUE (Wert 1), wenn die gegebene Bedingung zutrifft, und FALSE (Wert 0), wenn sie nicht zutrifft.

C hat keinen speziellen Datentyp für Vergleiche. Es werden einfach Integer verwendet, wobei 0 die Bedeutung von FALSE hat, jeder andere Wert bedeutet TRUE.

Die folgenden relationalen Operationen sind definiert:

>	größer als
>=	größer oder gleich
==	gleich
!=	ungleich
<=	kleiner oder gleich
<	kleiner als

Programmbeispiel:

```
#include <stdio.h>
void main()
{
    float a, b;
    printf(„\nGeben Sie zwei Zahlen ein:");
    scanf("%f %f", &a,&b);
    if (b == 0.0)       /* wenn b den Wert 0 hat
        printf(„Verhältnis nicht definiert!\n");
    else
        printf („Das Verhältnis ist %f  \n", a/b);
}
```

3.6.5 Logische Operatoren

Logische Operatoren dienen zur Verknüpfung logischer Werte:

&&	logisches UND (AND)
||	logisches ODER (OR)
!	logische Negation (NOT)

Beispiel:

```
    if (index < grenze && array[index] != 0) ...
```

Hier wird zuerst geprüft, ob der Wert der Variablen index kleiner als der Wert der Variablen grenze ist. Wenn diese Prüfung bereits FALSE ergibt, bricht C die Auswertung ab, weil das Ergebnis durch eine Prüfung weiterer Bedingungen nicht mehr beeinflußt werden kann.

3.6.6 Bitmanipulatoren

Bitmanipulationen sind in C über sechs verschiedene Operatoren möglich:

<<	Linksschieben
>>	Rechtsschieben
&	AND (bitweises UND)
|	OR (bitweises ODER)
^	XOR (bitweise Antivalenz)
~	NOT (bitweise Negation)

Programmbeispiel:

```
#include <stdio.h>
void main ()
{
     int a,b,c;
     char *format1, *format2;
     format1 = " %04X %s %04X  %04X\n";
     format2 = " %c%04X = %04X\n";
     a = 0x0FF0;    b = 0xFF00;

     c = a << 4;    printf(format1, a, „<<“, 4, c);
     c = a » 4;     printf(format1, a, „>>“, 4, c);
     c = a & b;     printf(format1, a, „&“, b, c);
     c = a | b;     printf(format1, a, „|“, b, c);
     c = a ^ b;     printf(format1, a, „^“, b, c);
     c = ~a;        printf(format2, „~“, a, c);
     c = -a;        printf(format2, „-“, a, c);
}
```

Definitionen für den Keil-C-Compiler:

>>:
Der Operand wird um eine Anzahl Stellen nach rechts geschoben. Ist der Operand vom Typ unsigned, so wird eine 0, ansonsten das Vorzeichen von links nachgezogen (arithmetic shift right). Bei Variablen mit negativen Vorzeichen wird der Wert nie größer -1, bei positivem Vorzeichen kann der Wert 0 erreicht werden.

<<:
Der Operand wird um eine Anzahl Stellen nach links geschoben. Von rechts werden Nullen nachgeschoben (man spricht hier auch von einer Multiplikation mit 2^{Anzahl}).

Für Rotatebefehle bietet das Header-File „**INTRINS.H“** geeignete Funktionen für die Datentypen char, int und long an.

3.6.7 Kombinatorische Operatoren

Alternativ zur sonst üblichen Form

a = a + b

erlaubt C die Abkürzung

a += b

mit jedem der bis jetzt besprochenen Operatoren. Sie schreiben also statt

<Variable> = <Variable> <Operator> <Ausdruck>,-

nur mehr

<Variable> <Operator>= <Ausdruck>;

Beispiele:

```
        a = a + b     wird verkürzt  zu      a += b;
        a = a - b     wird verkürzt  zu      a -= b;
        a = a * b     wird verkürzt  zu      a *= b;
        a = a / b     wird verkürzt  zu      a /= b;
        a = a % b     wird verkürzt  zu      a %= b;
        a = a >> b    wird verkürzt  zu      a »= b;
        a = a << b    wird verkürzt  zu      a «= b;
        a = a & b     wird verkürzt  zu      a &= b;
        a = a | b     wird verkürzt  zu      a |= b;
        a = a ^ b     wird verkürzt  zu      a ^= b;
```

3.6.8 Adress-Operatoren

In C gibt es zwei spezielle Operatoren für Zeiger und Adressen:

& ("Adresse von")
liefert die Adresse einer angegebenen Variablen zurück. Wenn summe eine Variable beliebigen Typs ist, dann liefert &summe jene Adresse zurück, ab der der Inhalt von summe gespeichert ist.

* ("Indirektion")
liefert den Inhalt der Speicherzelle zurück, auf die ein Zeiger weist. Wenn string ein Zeiger auf ein char-Array ist, dann liefert *string das erste Zeichen dieses Arrays.

Programmbeispiel:

```
#include <stdio.h>

vold main()

{
int summe;
char *string;
summe = 5 + 3;
string = „Anwendungsprogrammierung in C“;
printf("summe = %d  &summe = %p\n",summe,&summe);
printf("*string = %c  string = %p\n",*string, string);
}
```

3.6.9 Komplexe Ausdrücke

Zuordnungen können in Klammern gesetzt und als Ausdrücke betrachtet werden. Ein solcher Ausdruck hat denselben Wert wie die Variable links des Gleichheitszeichens:

(sum = 5 + 3)	ist identisch mit dem Wert 8
((sum=5+3) < 10)	ergibt folglich immer den Wert true

Beispiel:

```
if ((ch = getch()) ==‘q’)
     printf ("Auf Wiedersehen!\n");
else
     printf ("Sie haben %c eingegeben\n", ch);
```

Es wird zuerst die Funktion getch() aufgerufen, die auf die Eingabe eines Zeichens über die Tastatur wartet, dann dieses Zeichen der Variablen ch zugeordnet und schließlich wird geprüft, ob ch das Zeichen 'q' enthält. Wenn ja, verabschiedet sich das Programm; wenn nein, wird das in ch gespeicherte Zeichen ausgegeben.

⌦ **Kommas als Trennung von Operationen**

Innerhalb eines runden Klammerpaares können mehrere Operationen hinter einander gestellt werden, wenn sie durch Kommas voneinander getrennt sind. C wertet diese Operationen von links nach rechts aus und benutzt das Ergebnis der letzten Operation als Wert des gesamten Ausdrucks. Wenn ch_alt und ch den Typ char haben, dann ordnet der Ausdruck

(ch_alt = ch, ch = getch())

zuerst ch_alt den Wert von ch zu, liest dann via getch() ein Zeichen von der Tastatur, ordnet dieses Zeichen ch als neuen Wert zu und liefert es gleichzeitig als Ergebnis des Ausdrucks.

Das folgende Programmfragment liest zwei Zeichen hintereinander von der Tastatur und vergleicht sie miteinander:

```
if (chl = getch(), (ch2 = getch()) == chl)
     printf(„Sie haben zweimal %c getippt\n“, chl);

else
     printf(„Sie haben %c %c getippt\n“, chl, ch2);
```

3.7 Selektion

3.7.1 IF .. ELSE

Die grundlegende Struktur einer if-Anweisung sieht so aus:

```
if (Vergleich)
        Anweisung;
else
        Anweisung;
```

Beispiel:

```
if (chl = getch(), (ch2 = getch()) == chl)
     printf(„Sie haben zweimal %c getippt\n“, chl);

else
     printf(„Sie haben %c %c getippt\n“, chl, ch2);
```

3.7.2 SWITCH .. CASE

Die switch - case - Struktur entspricht einer Verzweigung:

Programmbeispiel:

```
/* ********************************************************* */
/* ****** Datei:    Rechner.C ******************************* */
/* ****** Datum:    10.05.1993      Prg.: Dr. Humer J.    ****** */
/* ****** Compiler: Keil - C 51 Version 3.40 ****************** */
/* ********************************************************* */


#pragma large           /* for 8051 only */
#pragma ROM (LARGE)     /* for 8051 only */
#include <reg552.h>     /* for 8051 only */
#include <stdio.h>

/*Variablendefinitionen*/
float zahl1,zahl2;
char op;
float erg;

/* Hauptprogramm */
main()
{
while(1)
        {
        printf("Geben Sie zwei Zahlen ein: \n");
        printf("Zahl1 Operator Zahl2 <return>, Komma mit Punkt\n");
        scanf("%f %c %f",&zahl1,&op,&zahl2);
        printf("Zahl 1   = %7.2f \n",zahl1);
        printf("Operator = %c\n",op);
        printf("Zahl 2   = %7.2f \n",zahl2);
        printf("-------------------------\n");
        switch(op)
        {
                case '+':
                        printf("Ergebnis = %7.2f\n",(float)zahl1+zahl2);
                        break;
                case '-':
                        printf("Ergebnis = %7.2f\n",(float)zahl1-zahl2);
                        break;
                case '/':
                        printf("Ergebnis = %7.2f\n",(float)zahl1/zahl2);
                        break;
                case '*':
                        printf("Ergebnis = %7.2f\n",(float)zahl1*zahl2);
                        break;
                default:
                        printf(„Nicht implementiert!“);
                        break;
        }  /* end switch */
        printf("\n");

        }  /* end while */

} /* end main */
```

⌧ Der **default - Zweig** wird ausgeführt, wenn keine **case - Marke** paßt.

- Die Reihenfolge der einzelnen Fälle ist unwichtig, es dürfen jedoch unter den case - Marken nicht zwei Einträge mit demselben Wert auftreten.

- Nach jeder Marke ist eine beliebige Anzahl von Anweisungen zulässig.

- Es dürfen mehrere case - Marken einer Anweisungsfolge zugeordnet werden.

- Das Weglassen einer break - Anweisung erzeugt einen Durchfluß, das heißt, es werden alle nachfolgenden Anweisungen bis zum nächsten break ohne weitere Prüfung der case - Marken ausgeführt.

Programmbeispiel: Erraten einer Zahl von 1 bis 15 in drei Versuchen

```
#include  <stdio.h>
#include  <conio.h>    /* PC only */
#define SUCCED 0

void main ()
{

    char line[80];  /* Eingabezeile */
    int found = 0;  /* hab ich sie gefunden? */
    int n = 3;      /* wieviele Versuche noch übrig */
    int range = 4;  /* Änderung fuer nächsten Versuch*/
    int try = 8;    /* nächster Versuch */
    char reply;     /* die Anwort des Benutzers */

    clrscr();
    printf("Antworten Sie nach jedem Versuch mit\n");
    printf("H, falls er zu hoch ist\n");
    printf("N, falls er zu niedrig ist\n");
    printf("E, falls ich die Zahl erraten habe.\n");

    while (n > 0 && !found)
    {
        printf("Ich tippe %d\n", try);
        scanf ("%s", line);
        reply = line[0];
        switch (reply)
        {
        case 'H':
        case 'h': try -= range;
                    range /= 2;  --n;
                    break;
        case 'N':
        case 'n': try += range;
                    range /= 2;  --n;
                    break;
        case 'E':
        case 'e':  found = 1;
                    break;
        default: printf("Bitte nur H, b" oder E.\n");
        } /*switch*/
     } /* while */
   printf ("'Ihre Zahl ist %d. \n", try)
   printf(,"Danke fuer das Spiel.\n,11);
} /*main */
```

3.8 Iteration mit Schleifen

Schleifen führen bestimmte Folgen von Befehlen wiederholt aus. C stellt drei mögliche Arten von Schleifen zur Verfügung, von denen zwei eigentlich nur ein Spezialfall der ersten sind.

3.8.1 WHILE - Schleife

Programmbeispiel:

```
#include <stdio.h>
#include <conio.h>      /* PC only */

void main()
{
     char *string;
     int wdh = 1;
     string = „Anwendungsprogrammierung in C";
     while (wdh <= 10)
     {
            printf(„Durchlauf %2d:  %s\n", wdh, string);
            wdh++;
     }
}
```

Die Schleife kann auch so programmiert werden:

```
     ....
     wdh = 0;
     while (wdh++ < 10)
            printf(„Durchlauf %2d: %s\n", wdh, string);
     .....
```

Ein weiteres Programmbeispiel:

```
#include <stdio.h>
#include <conio.h>      /* PC only */

void main()
{
     int ch;
     int len = 0; /* zählt die Anzahl der Eingaben */

     puts(„Geben Sie einen mit RETURN beendeten Satz ein.");

     while ((ch=getch()) != 13)     /* solange nicht return */
     {
            putch(ch);        /*eingegebenes Zeichen ausgeben */
            len++;            /* Längenzähler um 1 erhöhen */
     }
     printf(„\nIhr Satz ist %d Zeichen lang. \n", len);
     getch();
}
```

3.8.2 FOR - Schleife

Die formale Definition von der for - Schleife lautet:

```
     for (Ausdruck 1; Ausdruck 2; Ausdruck 3)
            Befehlsblock;
```

Diese Anweisung wird vom C-Compiler eigentlich folgend umgesetzt:

```
     Ausdruck 1;
     while (Ausdruck 2)
     {
            Befehl;
            .....
            Ausdruck 3;
     }
```

Programmbeispiel:

```
#include <stdio.h>

void main()
{
     char *string;
     int wdh;            /*Zähler für die Wiederholungen*/
     string = „Anwendungsprogrammierung in C“;
     for (wdh = 1; wdh <= 10; wdh++)
            printf(„Durchlauf %2d:  %s\n“, wdh, string);
}
```

3.8.3 DO .. WHILE Schleife

Die formale Definition von do ... while:

```
     do
      Befehl;
     while (Ausdruck);
```

Programmbeispiel:

```
#include <stdio.h>
#include <conio.h>      /* PC only */
void main()
{
     float a, b, division;
     char ch;

     do
     {
            printf(„\nGeben Sie zwei Zahlen ein:“);
            scanf(„%f %f, &a, &b);
            if (b==0.0);
                  printf(„Verhältnis ist nicht definiert!\n);
```

```
            else
            {
                  division = a/b;
                  printf(„Das Verhältnis ist %f \n“,division);
            } /* End if */
            printf(„X: Ende, jede andere Taste -> weiter\n“);
     }while ((ch = getch()) != ‘X’);
}/* End main */
```

3.9 Zusätzliche Steuerung des Programmablaufes

3.9.1 Befehl Return

Dieser Befehl wird in Funktionen verwendet. Wenn die Funktion einen Wert zurückliefern soll, dann muß der Befehl Return verwendet werden.

Programmbeispiel:

```
int max(int a, int b)
{
     if (a < b) return (a);
     else return (b);
}
```

3.9.2 Befehl Break

Dieser Befehl bricht nicht nur eine Ausführung von CASE ab, auch Schleifen können damit vorzeitig verlassen werden.

Programmbeispiel:

```
#include <stdio.h>
void main()
{
     int i, eintrag;
     int liste[10];
     printf(„\nAbbrechen mit negativer Zahl\n\n“);
     for (i=1; i<=10; i++)
     {[
            printf(„Eingabe des Eintrags %d:  “, i);
            scanf(„%d“, &eintrag);
            if (eintrag < 0) break;           /*negativ->Abbruch*/
            liste[i] = eintrag;
     }                                        /*End for */
}                                             /*End main*/
```

3.9.3 Befehl Continue

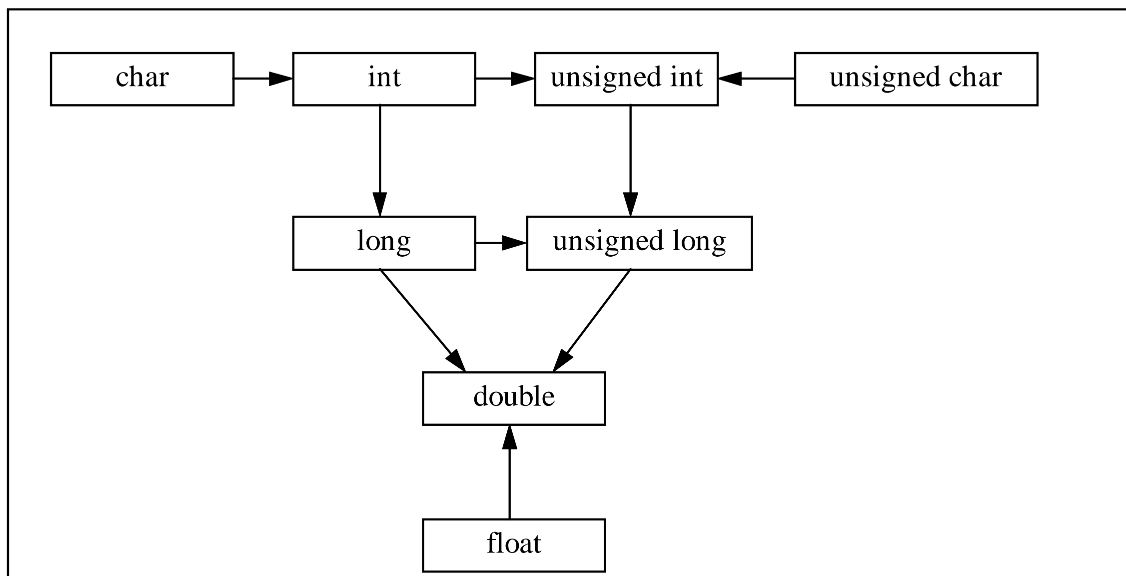
Dieser Befehl bewirkt das Gegenteil von BREAK: Während Break zum Ende einer Schleife springt und die Schleife abbricht, wiederholt continue eine Schleife sofort, springt also zu ihrem Anfang und der nächste Durchlauf startet sofort.

Programmbeispiel:

```
#include <stdio.h>
void main()
{
     int i, eintrag;
     int liste[10];

     printf(„\nAbbrechen mit negativer Zahl\n\n");
     i = 1;
     while (i <= 10)
     {
            printf(„Eingabe des Eintrags %d:",i);
            scanf(„%d", &eintrag);
            if (eintrag < 0) continue;
                        /* negativer Eintrag bewirkt eine Wiederholung*/
            liste[i] = eintrag;
            i++;
     }
}
```

3.10 Typkonvertierung

Unter dem Begriff Typkonvertierung versteht man die Umsetzung eines Datentyps in einen anderen. Treffen in einem Ausdruck Operanden verschiedener Datentypen aufeinander, so nimmt der C51-Compiler eine automatische Typenkonvertierung vor. Alle Operanden wandelt der Compiler vor der Berechnung in den größten Typ. In welcher Reihenfolge nun die Datentypen gewandelt werden, zeigt folgende Grafik.



3.10.1 Automatische Konvertierung

Beispiel:

```
char wert1 = 23;
int wert2 = 47;
float erg, wert3 = 5;

main()
{
     erg = wert1 + wert2 + wert3
                                    /* Ergebnis erg ist vom Typ float */
}
```

3.10.2 signed ⇔ unsigned

Werden Variablen vom Typ signed nach unsigned gewandelt, so ist darauf zu achten, daß keine negativen Werte gewandelt werden. Werden Werte vom Typ unsigned nach signed gewandelt, die den Wertebereich von signed überschreiten, so sind die Werte undefiniert.

Programmbeispiel:

```
char a;
unsigned char b;

main()
{
     a = -120;
     b = a;       /* b enthält den Wert 136 */

     b = 200;
     a = b;       / a enthält den Wert -56 */
}
```

3.10.3 char ⇒ int

Wird der Datentyp erweitert, so ist dies immer ohne Datenverlust möglich. Der Compiler legt bei der Expandierung eine Variable des geforderten Datentyps an, initialisiert diese mit Null und kopiert danach den zu expandierenden Wert in die Variable.

Programmbeispiel

```
char a;
unsigned char b;

int c;
unsigned int d;

main()
{
     a = -120;
     b = 200;
     d = a;       /* d enthält den Wert 65416 */
     c = a;       /* c enthält den Wert 200 */
}
```

3.10.4 int ⇒ float

Bei einer Verminderung des Formats kommt es unweigerlich zu einem Datenverlust. Dabei wird das MSB der Int-Variable bei der Typenkonvertierung außer acht gelassan. Die Vorzeichen können somit nicht übernommen werden.

Programmbeispiel:

```
char a;
unsigned char b;
int c;
unsigned int d;

main()
{
     c = -200;
     d = 2000;
     a = c; /* a enthält den Wert 56 */
     b = d; /* b enthält den Wert 208 */
}
```

Kommt es bei der Typkonvertierung zu einer Bereichsüberschreitung, ist das Ergebnis undefiniert.

3.10.5 int ⇒ float

Da die Mantisse des float-Typs 23 bit beträgt, ist eine Typumwandlung von Datentyp int und vom Datentyp unsigned int nach float ohne Datenverlust möglich. Da double denselben Wertebereich besitzt, ist auch hier eine Typkonvertierung ohne Datenverlust möglich.

3.10.6 long ⇒ float

Wird ein long-Wert in einen float-Wert umgewandelt, so kann es vorkommen, daß der Wert nicht exakt dargestellt werden kann. Das Resultat kann nun der nächstgrößere oder -kleinere darstellbare Wert sein.

Programmbeispiel:

```
long wert;
float erg;

main()
{
     wert = 5343433;
     erg = wert;  /* erg = 5.34234+E6 */

     wert = 53492349;
     erg = wert;  /* erg = 5.34934+E7 */

     wert = 21474835593;
     erg = wert;  /* erg = 2.14748+E9 */
}
```

3.10.7 float ⇒ int

Bei der Typumwandlung von float nach int wird der Dezimalbruch abgeschnitten, es findet keine Rundung statt. Ist bei der Typumwandlung der int-Wert nicht darstellbar, so ist das Ergebnis unbestimmt.

Beispiel:

```
int erg;
float wert;
main()
{
     wert = 534.23;
     erg = wert;        /* erg = 534 */
     wert = 534.98;
     erg = wert;        /* erg = 534 */
     wert = 40232.2
     erg = wert;        /* erg = -25304 (unbestimmt) */
     wert = -52334.2
     erg = wert;        /* erg = 13202 (unbestimmt) */
}
```

3.10.8 float ⇒ long

Wie auch bei der float -> int Konvertierung wird hier der Dezimalbruch abgeschnitten. Ist bei der Typumwandlung der long-Wert nicht darstellbar, so ist auch hier das Ergebnis unbestimmt.

3.10.9 cast - Operator

Der cast-Operator erzwingt eine Typkonvertierung. Dies ist z.B. dann sinnvoll, wenn ein Ausdruck ein Ergebnis liefern kann, das die Bereiche der eingesetzten Datentypen überschreitet.

Beispiel:

```
a = (int)b;             /* Typkonvertierung einer Variablen */
b = (long)eingabe(10,10);
            /* Das Ergebnis der Funktion eingabe wird konvertiert */
```

Beispiel:

```
/* Umwandlung mittels cast-Operator */

unsigned char wert1, wert2;
int erg;

main()
{
     wert1 = 230;
     wert2 = 140;
     erg = wert1 + wert2;           /* Ergebnis = 114 */
     erg = (int)wert1 + wert2;      /* Ergebnis = 370 */
}
```

4 C51-Datentypen

In C51 müssen alle Datentypen definiert bzw. deklariert sein, bevor sie verwendet werden können.

☞ bit

Verwendung für den Datentyp BIT.

☞ char

Datenlänge 8 bit
umfaßt ein einzelnes Zeichen und benötigt ein Byte Speicherplatz
Wertebereich: -128 ... 0 ... +127
unsigned char: 0 ... 255

☞ int

Datenlänge: 2 Byte
zur Speicherung ganzer Zahlen,
Wertebereich: -32786 ... 0 ... 32767
unsigned int: 0 ... 65536

☞ long

Datenlänge: 4 Byte
Ist ein Integertyp mit 4 Byte Länge
Wertebereich: -2147483648 ... 0 ... 2147483647

☞ float

Datenlänge: 4 Byte
für rationale Zahlen mit einfacher Genauigkeit. 7 signifikante Ziffern !
Wertebereich: 3,4 E-38 ... 0 ... 3,4 E+38

☞ double

Datenlänge: 8 Byte
für rationale Zahlen mit erhöhter Genauigkeit. 16 signifikante Ziffern !
Wertebereich: 1,7 E-308 ... 0 ... 1,7 E+308

Datentyp	8-bit-Rechner (805X)	16-bit-Rechner	32-bit-Rechner
bit	1 bit	nicht vorhanden	nicht vorhanden
char	8 bit	8 bit	8 bit
int	16 bit	16 bit	16 bit
short	16 bit	16 bit	32 bit
long	32 bit	32 bit	32 bit
float	32 bit	32 bit	32 bit
double	32 bit	64 bit	64 bit

Tab. Vergleich der Datentypen bei unterschiedlichen Prozessoren

4.1 Globale Variablen

Werden Variablen außerhalb einer Funktion definiert oder deklariert, so spricht man von globalen Variablen. Diese Variablen können von jeder Funktion im aktuellen Modul verwendet werden.

Globale Variablen sind statische Variablen und haben einen festen Speicherplatz.

Beispiel:

```
int a;                      /* Definition der globalen Variablen */
void schleifel() {          /* Beginn der Funktion schleife 1 */
       a = 1
                            /* Ende der Funktion schleife 1 */
}
int b;                      /* Definition der globalen Variablen */
void schleife2() {          /* Beginn der Funktion schleife 2 */
       b = a + 2;
}                           /* Ende der Funktion schleife 2 */
```

Während die Variable a in beiden Funktionen gültig ist, beschränkt sich der Gültigkeitsbereich der Variablen b nur auf die Funktion schleife 2

4.2 Lokale Variablen

Sie werden innerhalb einer Funktion definiert, dbaie spricht man auch von lokalen Variablen. Lokale Variablen benötigen nur dann einen festen Speicherplatz, wenn sie mit der Speicherklasse static definiert wurden.

☞ Lokale Variablen sind dynamische Variablen und haben keinen festen Speicherplatz.

Beispiel:

```
void test() {               /* Beginn der Funktion test */
       char a;              /* Definition der lokalen Variablen */

       a = 1;
         {                  /* Beginn einen geschachtelten Blocks */
         char b;            /* Definition einer Variablen */

         b = a;             /* Wertzuweisung */
         }                  /* Ende einen geschachtelten Blocks */
}                           /* Ende der Funktion test */
```

Während die Variable a in der gesamten Funktion test gültig ist, ist die Variable b nur im verschachtelten Block gültig.

```
  void test() {                /* Beginn der Funktion test */

   char a, b;           /* Definition von lokalen Variablen
        a = 4;          /* Wertzuweisung  ❶ */
        b = 3;
         {              /* Beginn einen vorschachtalten Blocks */
         char a;        /* Definition von lokalen Variablen */
         a = 35;        /* Wertzuweisung ❷ */
           }           /* Ende eines verschachtelten Blocks */
         a = a + b;
      }
```

Im letzten Beispiel wurde die Variable a zweimal definiert. Das erste Mal in der Funktion Test, das zweite Mal im verschachtelten Block. Für den Fall, daß ein Variablenname mehrmals verwendet wird, gilt eine besondere Regel:

☞ Es gilt: Eine Variable ist immer in dem Block gültig, in dem sie definiert wurde.

Dies hat nun folgende Auswirkung: Im verschatelten Block wird eine neue Variable mit dem Name a definiert. Diese hat keinen Einfluß auf die Variable a in der übergeordneten Funktion. Bei der Wertzuweisung ❷ verändert sich nur der Wert der aktuellen Variablen. Wird der verschachtelte Block verlassen, so enthält die Variable a immer noch den Wert aus der Wertzuweisung ❶ (den Wert 4).

4.3 Speicherklassen

4.3.1 auto

Die Variable ist nur im jeweiligen Funktionsblock gültig

Beispiel:

```
void schleife() {          /* Beginn des Funktionsblocks */
      auto char a, b;
         ...;
         {                 /* Beginn des Schachtelblocks */
           auto char c, d;
           ...;
         }                  /* Ende des Schachtelblocks */
         ...;
}                           /* Ende des Funktionsblocks */
```

4.3.2 static

Wird einer globalen Variablen die Speicherklasse static zugewiesen, so ist die Variable nur im aktuellen Modul gültig. Weist man einer lokalen Variablen die Speicherklasse static zu, so wird die lokale Variable nach verlassen der Funktion oder Beendigung eines Blocks nicht von anderen lokalen Variablen überlagert. Bei einem erneuten Eintritt in den Block ist der alte Inhalt noch erhalten.

```
static a;          /* Die Variable a ist nur im Modul gültig */
  test()  {
        a = 0;
        ...;
}

  erg()  {
     static b = 0;   /* Die Variable b wird vor dem ersten Eintritt in */
        b = a+3;     /* die Funktion mit dem Wert 0 vorbelegt */
}
```

4.3.3 extern

Die Speicherklasse extern bewirkt, daß dem Compiler Funktionen bzw. Variablen erklärt (deklariert) werden, die in einem anderen Modul definiert wurden. Die extern-Anweisung kann nur bei globalen Variablen verwendet werden.

```
  int  a;

  void schleife1() {          /* Beginn der Funktion */
        extern int b;
        a = 2;
        b = 3;
     ...;
 }                           /* Ende den ersten Blocks */

  int  b;
  void schleife2() {          /* Beginn der Funktion */
        b = a + b;
            ... ;
 }                            /* Ende den ersten Blocks */
```

4.3.4 register

Der C51-Compiler versucht in der Optimierungsstufe 4, Variablen immer in Registern abzulegen. Welche Varible̲n nun in Registern abglegt werden, kann man mit der Speicherklasse register steuern.

```
            #pragma   small  /* ❺ */
            #pragma code ot(4)

            t_small() {            /* Funktion t_small */
           register int a, b, c, erg;

           a = 3;
           b = 4;
           c = a  +  b;
           erg = c + a + b;


 Listfile von Funktion t-Small

                 FUNCTION t_small (BEGIN)
```

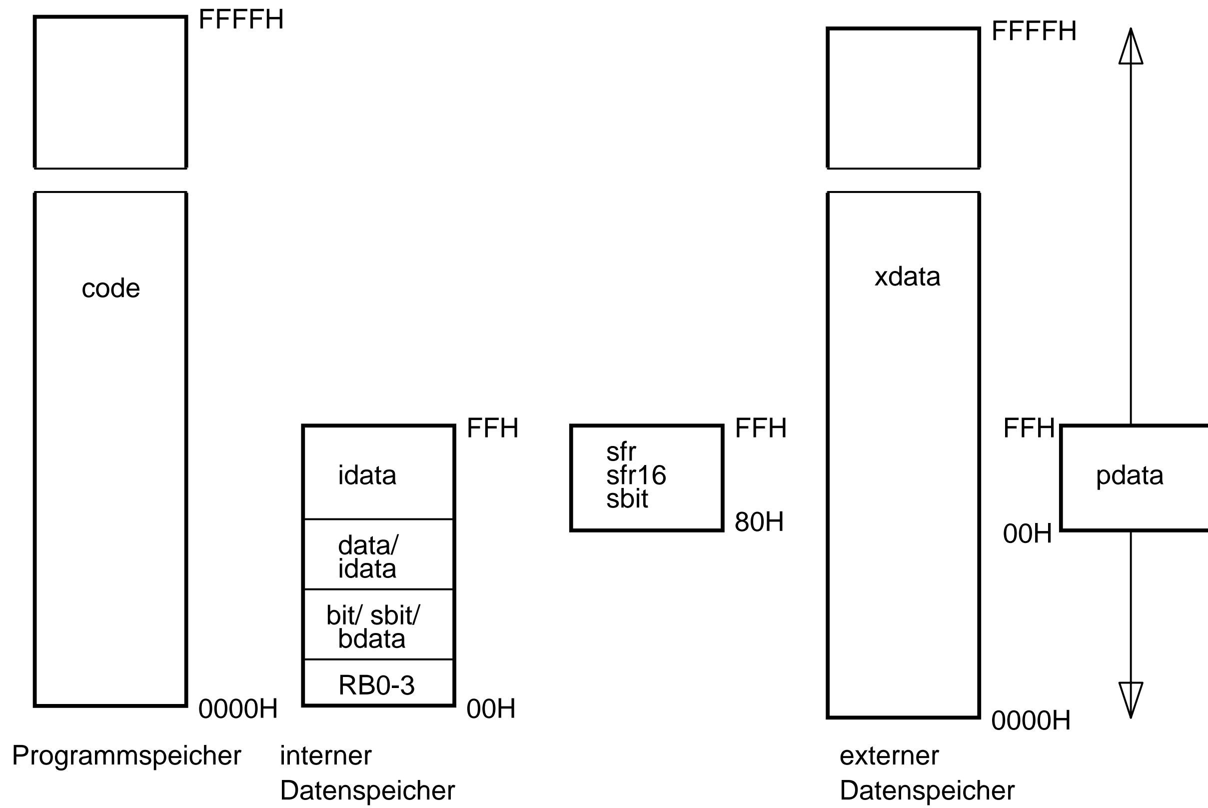
```
                                                    ; SOURCE LINE    # 7
;---- Variable a assigned to Register 'R6/R7'          ❶
 7E03          mov      R7,#03H
 7E00          mov      R6,#00H
                                                    ; SOURCE LINE    # 8
;---- Variable b assigned to Register 'R4/R5'          ❷
 7D04          mov      R5,#04H
 7C00          mov      R4,#00H
                                                    ; SOURCE LINE    # 10
 EF            mov      A,R7
 2D            ADD      A,R5
000A   F500      R     mov       C+01H,A               ❸
000C   EE              mov       A,R6
000D   33              RLC       A
000E   F500      R     mov       C,A                   ❹

 ....        ........            ...
001F   22              RET
```

4.4 Speichertypen

Jede Variable wird entweder im internen oder im externen RAM-Bereich abgelegt. Mit dem Speichertyp wird angegeben, in welchem Speicherbereich die jeweilige Variable abgelegt werden soll. Wird kein Speichertyp angegeben, so wird der Speichertyp durch das Speichermodell festgelegt (Compileranweisung).

Beispiel für 8-bit Contoller 805X

Speicherbereich	Adresse	Speichertyp
Direkt oder indirekt adressierbarer Bereich im internen Datenspeicher	0-7FH	**data**
Adressierbare Bits im internen Datenspeicher	20-2FH	**bit/sbit**
Adressierbare Bits im SFR-Bereich	80-FFH	**sbit**
Bitadressierbare Bytes im internen Datenspeiche	20-2FH	**bdata**
indirekt adressierbarer interner Datenspeicherbereich	0-7FH 80-FFH	**idata**
256 Byte externer Datenspeicherbereich	0-FFH	**pdata**
max. 64 Kbyte externes RAM	0-FFFFH	**xdata**
Code-Speicherbereich	0-FFFFH	**code**



Graphische Darstellung der einzelnen Speichertypen

BULME GRAZ

5 Handhabung des C51 - Compilers

5.1 Aufruf

Compileraufruf:

Beispiel:
c:\work>**C51 <filename>.c51**
Erklärung:

C51	Programmaufruf
<filename>.c51	Quelldatei in der Programmsprache „C“

Linkeraufruf:

Beispiel:
c:\work>**L51 <filename>.obj code(8000H) xdata(0C000H) RAMSIZE(256)**
Erklärung:

L51	Programmaufruf Linker - Keil
<filename>.obj	Objektdatei
RAMSIZE(256)	Interner Speicher 256 BYTE
code(8000h)	Programm soll ab 8000HEX gespeichert werden
xdata(0C000H)	Externer Arbeitsspeicher, beginnend mit ..

5.2 Steurerparameter:

Die Steuerparameter beeinflussen den Compiler beim Erstellen des Object-Codes. So besteht mit den Steuerparametern z.B. die Möglichkeit, alle Interrupts bei einem Funktionsaufruf zu sperren oder den erzeugten Programmcode auf Zeit oder Geschwindigkeit zu optimieren.

Syntax: #pragma (Steueranweisung)

Name	Abkürzung	Voreinstellung	#pragma	ext Steuerparameter
SAVE	—		*	
RESTORE	—	—	*	
DISABLE	—	—	*	
EJECT	EJ	—	*	
OPTIMIZE	OT	OT(5,Size)	*	*
[NO]REGPARMS	[NO]RP	RP		*
[NO]AREGS	[NO]AR	AR		*
REGISTERBANK	RB	RB(0)		*

Bild 15-1: Allgemeine Parameter * Parameter zulässig

Name	Abkürzung	Voreinstellung	#pragma	ext Steuerparameter
DEFINE	DF	–	*	*
[NO]EXTEND	–	EXTEND	*	*
[NO]LISTINCLUDE	[NO]LC	NOLC	*	*
[NO]SYMBOLS	[NO]SB	NOSB	*	*
[NO]PREPRINT	[NO]PP	NOPP		*
[NO]CODE	[NO]CD	NOCD	*	*
[NO]PRINT	[NO]PR	PR(Dateiname.LST)	*	*
[NO]COND	[NO]CO	COND	*	*
PAGELENGTH	PL	PL(69)	*	*
PAGEWIDTH	PW	PW(132)	*	*
[NO]DEBUG	[NO]DB	NODB	*	*
[NO]OBJECT	[NO]OJ	OJ(Dateiname.OBJ)	*	*
SMALL	SM	SM	*	*
COMPACT	CP	–	*	*
LARGE	LA	–	*	*
[NO]INTVECTOR	[NO]IV	IV	*	*
OBJECTEXTEND	OE	–	*	*
ROM	–	ROM(LARGE)	*	*
SRC**	–	SRC(Dateiname.SRC)	*	*

Bild 15-2: Primäre Parameter * Parameter zulässig ** Den Steuerparameter SRC gibt es erst ab Version 3.20

Achtung: Primäre Parameter müssen am Anfang des C-Moduls, noch vor den #include-Anweisung stehen.

Im weiteren werden nur die wichtigsten Steuerparameter beschrieben, eine ganaue Aufstellung findet sich im Handbuch.

5.2.1 SRC

Mit der Anweisung SRC besteht seit der Version 3.20 die Möglichkeit, alternativ zur OBJ-Datei eine Source-Datei zu erzeugen. Die so erzeugte Datei kann mit dem A51-Assembler übersetzt werden.

Syntax: #pragma SRC (Dateiname)

5.2.2 Optimize

Mit der Optimize Anweisung wird dem Compiler die Art der Optimierung angegeben. In der Optimize Anweisung sind folgende Einstellungen möglich.

Syntax: #pragma OPTIMIZE (0 bis 5, SPEED/SIZE)

5.2.3 Define

Mit der Define Anweisung wird Ihnen die Möglichkeit gegeben, den Preprozessor zu steuern. Der Preprozessor ist ein für sich eigenständiger Programmteil, der vor der Compilierung abläuft. Sie können mehrere, durch Leerzeichen getrennte, Argumente angeben.

Beispiel:

```
#define FOREVER for(;;) /* Endlosschleife */
```

5.2.4 Symbols (SB)

Bei dieser Angabe werden alle verwendeten Symbole in das Listing mit aufgenommen. Für jedes verwendete Objekt in die Speicherklasse, der Speichertyp, der Offset und die Größe des Objekts angegeben.

5.2.5 Code

CODE weist den C51-Compiler an, die erzeugten Assemblerbefehle in das Listing mit aufzunehmen.

5.2.6 DEBUG

Symbole und Zeilennummern werden in die Objektdatei mit aufgenommen. Diese Debug-Informationen sind zum Testen von Programmen notwendig, wenn mit dem Hochsprachendebugger gearbeitet werden soll.

5.2.7 Speichermodelle

Mit dem Speichermodell wird festgelegt, in welchem Speicherbereich sich die globalen und lokalen Variablen befinden, wenn kein Speichertyp angegeben wird.

5.2.7.1 small

Alle Variablen und lokalen Datensegmente von Funktionen werden in den data Bereich gelegt.
Der Zugriff auf dies Variablen ist sehr schnell. Es ist aber zu beachten, daß sich der Stack auch im internen Datenspeicher befindet.

5.2.7.2 compact

Alle Variablen und lokalen Variablen werden im externen Datenspeicher abgelegt. Der externe Datenspeicher ist hier auf ein PAGE (256Byte) begrenzt. Damit ist eine kurze Adressierungsform über @ R0/R1 möglich.

5.2.7.3 large

Alle Variablen und lokalen Datensegmente werden im externen Datenspeicher abgelegt. Der Datenspeicher kann bis zu 64 KB betragen. Die Datenadressierung kann nur über den DPTR erfolgen.

5.2.8 INTVECTOR (IV)

Für die Interruptfunktionen generiert der C51-Compiler einen 3 Byte Sprung LJMP. Der Vector liegt an der absoluten Adresse 8*n+3, wobei für n die Interruptnummer eingesetzt wird. Wird NOIV angewählt, so werden keine Interruptvektoren generieren. Damit ist die Möglichkeit gegeben, die Interruptvektoren in einem Assemblerprogramm bereitzustellen oder das Interruptprogramm direkt am Vektor abzulegen.

Literaturquellen:

MC-Tools 7, Der Keil-C51-Compiler, Einführung und Praxis, Michael Baldischweiler, Verlag: Feger & Co