

1. Einführung in die Programmiersprache C ☒
2. Gültigkeitsbereiche, komplexe Datentypen ☒
- 3. Kontrollstrukturen, Ein- und Ausgabe**
4. Zeiger, Felder und Zeichenketten
5. Makros, C-Entwicklungswerkzeuge
6. Dateisystem
7. Ausgewählte Beispiele
(Prozesse, Threads, ...)

- Verzweigung
 - if – else
 - switch - case
- Schleifen
 - while
 - do - while
 - for
- Sprünge
 - goto (wird z.B. im Linux Kernel verwendet)

Kontrollstrukturen

if – else - Verzweigung

- Syntax:

```
if( <Bedingung> )  
{  
    <Anweisung für Bedingung == TRUE>  
}  
else  
{  
    <Anweisung für Bedingung == FALSE>  
}
```

- Die else-Anweisung kann entfallen
- if-else-Anweisungen können geschachtelt werden
- <Bedingung>: beliebiger **Ausdruck**, der einen Wert zurück liefert.
„0“ wird als „FALSE“ gewertet, alle anderen Werte entsprechen „TRUE“
- Alternative: Auswahloperator

Syntax:

```
<Bedingung>?<AusdruckFuerTrue>:<AusdruckFuerFalse>
```

```
int main()
{
    float x=3.0, min=0.0, max=10.0;

    if(x > max)
    {
        printf("x ist zu gross\n");
    }
    else if(x < min)
    {
        printf("x ist zu klein\n");
    }
    else
    {
        printf("x liegt innerhalb \
              der Grenzen\n");
    }
}
```

```
if (x > y)
    max = x;
else
    max = y;
```

Alternative:

```
max = ((x>y) ? x : y);
```

Kontrollstrukturen

Beispiele if – else

3. Kontrollstrukturen, Ein-und Ausgabe

Ein potentieller Fehler ?!

```
int main()
{
    int a, b=1;

    if (a=b)
        printf(„Lösung 1\n");
    else
        printf(„Lösung 2\n");
}
```

Kontrollstrukturen

switch – case - Verzweigung

- Syntax:

```
switch( <Ausdruck> )  
{  
    case <const1>:  
        <Anweisung falls Ausdruck == const1>  
        break;  
    case <const2>:  
        <Anweisung falls Ausdruck == const2>  
        break;  
    ...  
    default:  
        <Anweisung falls keine Bedingung erfuehlt ist>  
}
```

- Es wird zu der **Anweisung** verzweigt, bei der der **<Ausdruck>** mit der **case**-Konstanten übereinstimmt
- Die **break**-Anweisung erzwingt das Verlassen des switch-Blockes. Eine fehlende **break** Anweisung bewirkt, dass alle nachfolgenden Anweisungen ebenfalls ausgeführt werden.


Kontrollstrukturen

Beispiel switch – case - Verzweigung

```
int main()
{
    int retval;

    retval = functionCall();

    switch(retval)
    {
        case -1:
            printf(„Fatal error\n“);
            break;
        case -2:
            printf(„Warning\n“);
            break;
        case -3:
            printf(„Error code: %d\n“, retval);
        default:
            printf(„Unknown error\n“);
    }
}
```



Kontrollstrukturen

while - Schleife

- Syntax:

```
while( <Bedingung> )  
{  
    <Anweisung>  
}
```

- Die **Anweisung** wird solange wiederholt ausgeführt, wie die **Bedingung** erfüllt ist
- Die Schleife kann mit einer **break**-Anweisung unmittelbar verlassen werden
- Mit einer **continue**-Anweisung wird sofort mit der nächsten Wiederholung begonnen

Kontrollstrukturen

while - Schleife

3. Kontrollstrukturen, Ein-und Ausgabe

```
int main()
{
    int zaehler = 0;

    while(1) /* Endlosschleife */
    {
        printf("Zaehler = %d\n", zaehler);
        if(zaehler >= 9) {
            break;
        }
        zaehler++;
    }
    zaehler = 0;

    while(zaehler < 10)
    {
        zaehler++;
        if(zaehler%2 == 0) {
            continue;
        }
        printf("Zaehler = %d\n", zaehler);
    }
    return 0;
}
```

Kontrollstrukturen

do – while - Schleife

- Syntax:

```
do
{
    <Anweisung>
}
while( <Bedingung> );
```

- Die **Anweisung** wird **mindestens einmal** ausgeführt und anschließend solange wiederholt, wie die **Bedingung** erfüllt ist
- Die Schleife kann mit einer **break**-Anweisung unmittelbar verlassen werden
- Mit einer **continue**-Anweisung wird sofort mit der nächsten Wiederholung begonnen

Kontrollstrukturen

Beispiel do – while - Schleife

```
int main()
{
    int zaehler = 0;

    do
    {
        printf("Zaehler = %d\n",zaehler);
        zaehler++;
    }
    while(zaehler < 10);
    return 0;
}
```

- Syntax:

```
for (<Initialisierung>; <Bedingung>; <Aktualisierung> )  
{  
    <Anweisung>  
}
```

- Zunächst wird die **Initialisierung** ausgeführt
- Solange die **Bedingung** erfüllt ist, wird die **Schleifenanweisung** durchgeführt
- Nach jeder Schleife wird (vor der Prüfung der **Bedingung**) die **Aktualisierung** ausgeführt
- Die Schleife kann mit einer **break**-Anweisung unmittelbar verlassen werden
- Mit einer **continue**-Anweisung wird sofort mit der nächsten Wiederholung begonnen

Kontrollstrukturen

Beispiel for - Schleife

```
int main()
{
    int i;

    for(i=0; i<10; i++)
    {
        printf("Zaehler = %d\n", i);
    }
    return 0;
}
```

Ist äquivalent zu:

```
int main()
{
    int i;

    i = 0;
    while(i < 10)
    {
        printf("Zaehler = %d\n", i);
        i++;
    }
    return 0;
}
```

Ein- und Ausgabe

- Für die Programmiersprache C existieren eine Vielzahl von Bibliotheken mit Funktionen für unterschiedlichste Anwendungsbereiche
- Ein Bibliothekspaket besteht „typischerweise“ aus folgenden Bestandteilen:
 - ❖ Datei, die den ausführbaren Binärcode der Bibliothek enthält
 - ❖ Header-Datei mit den Funktionsprototypen
 - ❖ Beschreibung der Bibliotheks-Funktionen, bestehend aus
 - Name und Beschreibung der Funktion
 - Übergabeparameter: Typ und Bedeutung
 - Mögliche Rückgabewerte und Ihre Bedeutung

Beim Aufruf von Bibliotheks- und eigenen Funktionen sollte der Rückgabewert (sofern vorhanden) überprüft werden!!!

```
double kehrwert(unsigned int value)
{
    if (value != 0)
        return (1.0/(double) value);
    else
        return -1.0;
}
```

```
int main()
{
    double retval;

    retval = kehrwert (0);
    if (retval < 0)
    {
        printf("Fatal error\n");
        return -1;
    }
    return 0;
}
```


- Viele Programme beinhalten eine Interaktion mit dem Benutzer ⇔ Ausgaben auf ein Anzeigegerät, Eingaben via Tastatur o.ä.
- Die C Standard Bibliothek bietet u.a. elementare Funktionen für die Ein- und Ausgabe von Daten von/auf den Geräten `stdin` (typischerweise die Tastatur) und `stdout` (typischerweise der Bildschirm)
- Die entsprechenden Funktions-Prototypen sind in der Datei `stdio.h` enthalten und werden die folgende Anweisung in eine Quelltextdatei eingefügt:
`#include <stdio.h>`
- Einfache Ein-/Ausgabe Funktionen
 - Standard-Ausgabe: `printf()`
 - Standard-Eingabe: `scanf()`

Einfache Ein-/Ausgabe

Standard-Ausgabe

- Syntax:
`int printf(const char *format,...);`
- Der Parameter `format` ist eine Zeichenkette, die typischerweise in folgender Form angegeben wird: **"ein Text mit Platzhaltern und ggf. Sonderzeichen"**
- `printf()` kann mit ,beliebiger' Anzahl von Argumenten (Aufzählung als komma-separierte Liste) nach `format` aufgerufen werden
- Platzhalter werden mit dem Prozentzeichen '%' eingeleitet und beinhalten eine Typangabe. Diese muss mit dem Typ eines entsprechenden Argumentes der Argumentliste (als geordnete Liste) übereinstimmen

Beispiele:

<code>%c</code>	-	<code>char</code>
<code>%s</code>	-	<code>string</code>
<code>%d</code>	-	<code>int</code>
<code>%f</code>	-	<code>float</code>
<code>%lf</code>	-	<code>double</code>

Einfache Ein-/Ausgabe

Standard-Ausgabe

- Rückgabewert:
 Fehlerfreie Ausführung: Anzahl der ausgegebenen Zeichen
 Im Fehlerfall: negativer Wert
- Ausgabe von Sonderzeichen durch Escape-Sequenz:
 Durch vorangestelltes ' \ ' wird das nächste Zeichen als Sonderzeichen gewertet:
 Beispiel:

\n	-	Zeilenumbruch
\r	-	Zeilenrücklauf
\t	-	Tabulator
- Sollen die Zeichen ' % ' oder ' \ ' selbst ausgegeben werden, so müssen sie doppelt angegeben werden

Einfache Ein-/Ausgabe

Beispiel Standard-Ausgabe

```
#include <stdio.h>
int main()
{
    char    kennung    = 'S';
    char    *titel     = „Programmierung in C“; // eine Zeichenkette
    int     nummer     = 10;
    int     jahr       = 2020;
    int     retVal;

    retVal = printf("Dies ist die %d.-Vorlesung \"%s\" im %cS-%d\n", \
        nummer, \
        titel, \
        kennung, \
        jahr);

    if (retVal < 0)
        return -1;

    return 0;
}
```

Einfache Ein-/Ausgabe

Standard-Eingabe

Für die Standard-Eingabe stehen folgende Funktionen zur Verfügung:

`int getchar(void)`

Liest ein Zeichen von der Tastatur und gibt dessen ASCII-Code zurück

`char *gets(char *s)`

Liest eine komplette Zeichenkette von der Standard-Eingabe. Die Funktion wird beendet sobald die Eingabe mit einem Carriage-Return abgeschlossen wurde

`int scanf(const char *format,...)`

Liest eine Zeichenkette von der Standard-Eingabe und wertet diesen gemäß der Formatierung aus

Einfache Ein-/Ausgabe

scanf()

- Syntax:

```
int scanf(const char *format,...);
```

- `scanf()` kann mit einer ‚beliebiger‘ Anzahl von Argumenten (komma-separierte Liste nach dem `format`-Parameter aufgerufen werden.

- Spezifikation des Parameters `format`:

Damit `scanf()` die Eingabe auswerten kann, muss die erwartete Formatierung angegeben werden. Dies geschieht durch Formatelemente, die mit dem Zeichen `'%'` eingeleitet werden, dem wiederum eine Typangabe folgt.

Beispiele:

<code>%c</code>	–	<code>char</code>
<code>%s</code>	–	<code>string</code>
<code>%d</code>	–	<code>int</code>
<code>%f</code>	–	<code>float</code>
<code>%lf</code>	–	<code>double</code>

Einfache Ein-/Ausgabe

Standard-Eingabe mit scanf()

- Die der Formatierung folgenden Argumente müssen **Adressen** von Variablen (wird im nächsten Kapitel besprochen ↗ Zeiger) sein, in denen die Eingabewerte abgelegt werden können
- Rückgabewert:
 Fehlerfreie Ausführung: Anzahl der erfolgreich eingelesenen Werte
 Im Fehlerfall:
 0 (kein fataler Fehler)
 oder
 EOF (diese Konstante ist in der Datei [stdio.h](#) definiert)

„These function returns the number of input items assigned. This can be fewer than provided for, or even zero, in the event of a matching failure. Zero indicates that, although there was input available, no conversions were assigned; typically this is due to an invalid input character, such as an alphabetic character for a '%d' conversion. The value EOF is returned if an input failure occurs before any conversion such as an end- of-file occurs. If an error or end-of-file occurs after conversion has begun, the number of conversions which were successfully completed is returned.” (Quelle: <http://www.manpagez.com/man/3/scanf>)

Einfache Ein-/Ausgabe

Beispiel Standard-Eingabe

```
#include<stdio.h>

int main()
{
    char answer;

    printf("Format C: ?\n");
    do
    {
        printf("\nEingabe: ");
        while (scanf("%c", &answer) == 0)
        {
            fflush(stdin); //Tastaturpuffer leeren
            printf("\nNeue Eingabe: ");
        }
        fflush(stdin); //Tastaturpuffer leeren (CR entfernen)

        printf("\nAnswer was: %c [%d]", answer, answer);

    } while ((answer != 'Y') && (answer != 'y'));

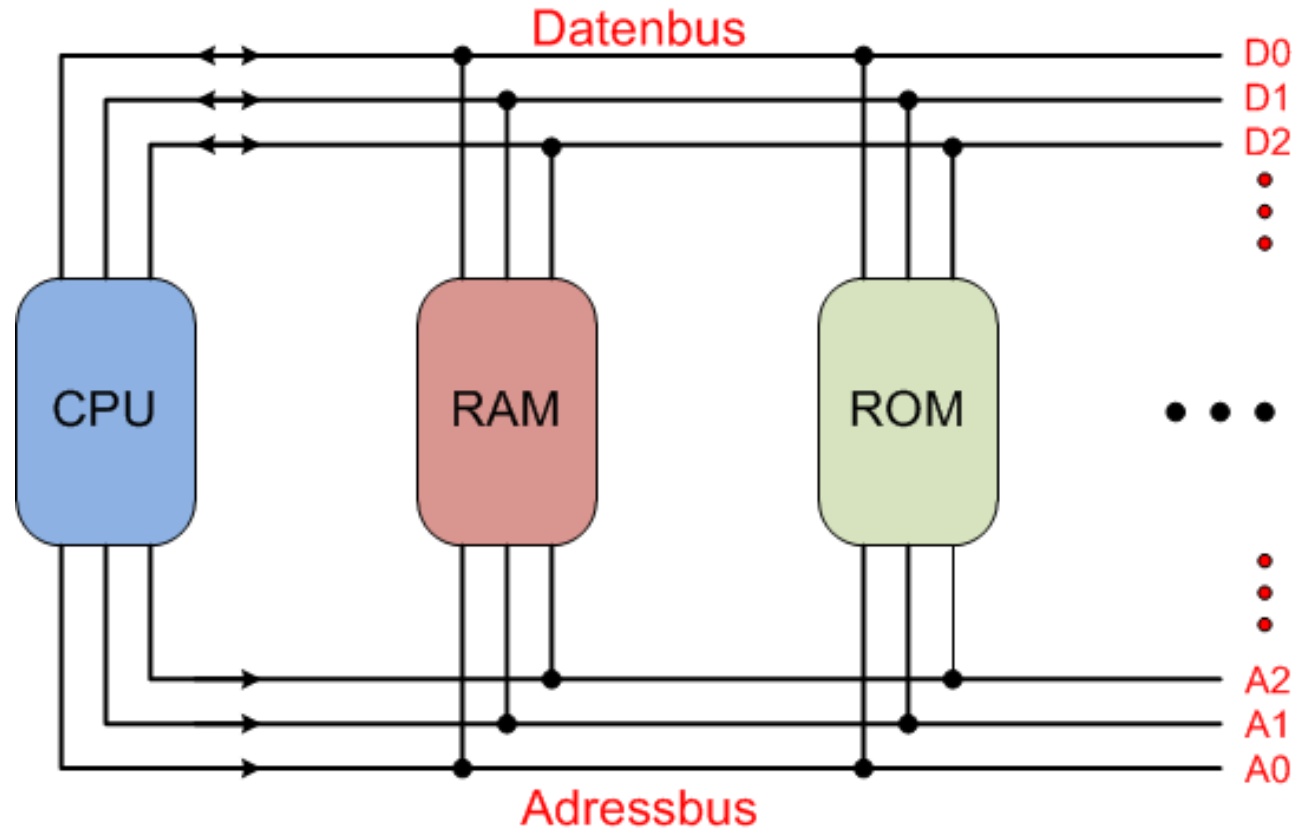
    return 0;
}
```


1. Einführung in die Programmiersprache C ☒
2. Gültigkeitsbereiche, komplexe Datentypen ☒
3. Kontrollstrukturen, Ein- und Ausgabe ☒
- 4. Zeiger, Felder und Zeichenketten**
5. Makros, C-Entwicklungswerkzeuge
6. Dateisystem
7. Ausgewählte Beispiele
(Prozesse, Threads, ...)

Vorbemerkung

Vereinfachte Rechnerstruktur

4. Zeiger, Felder und
Zeichenketten



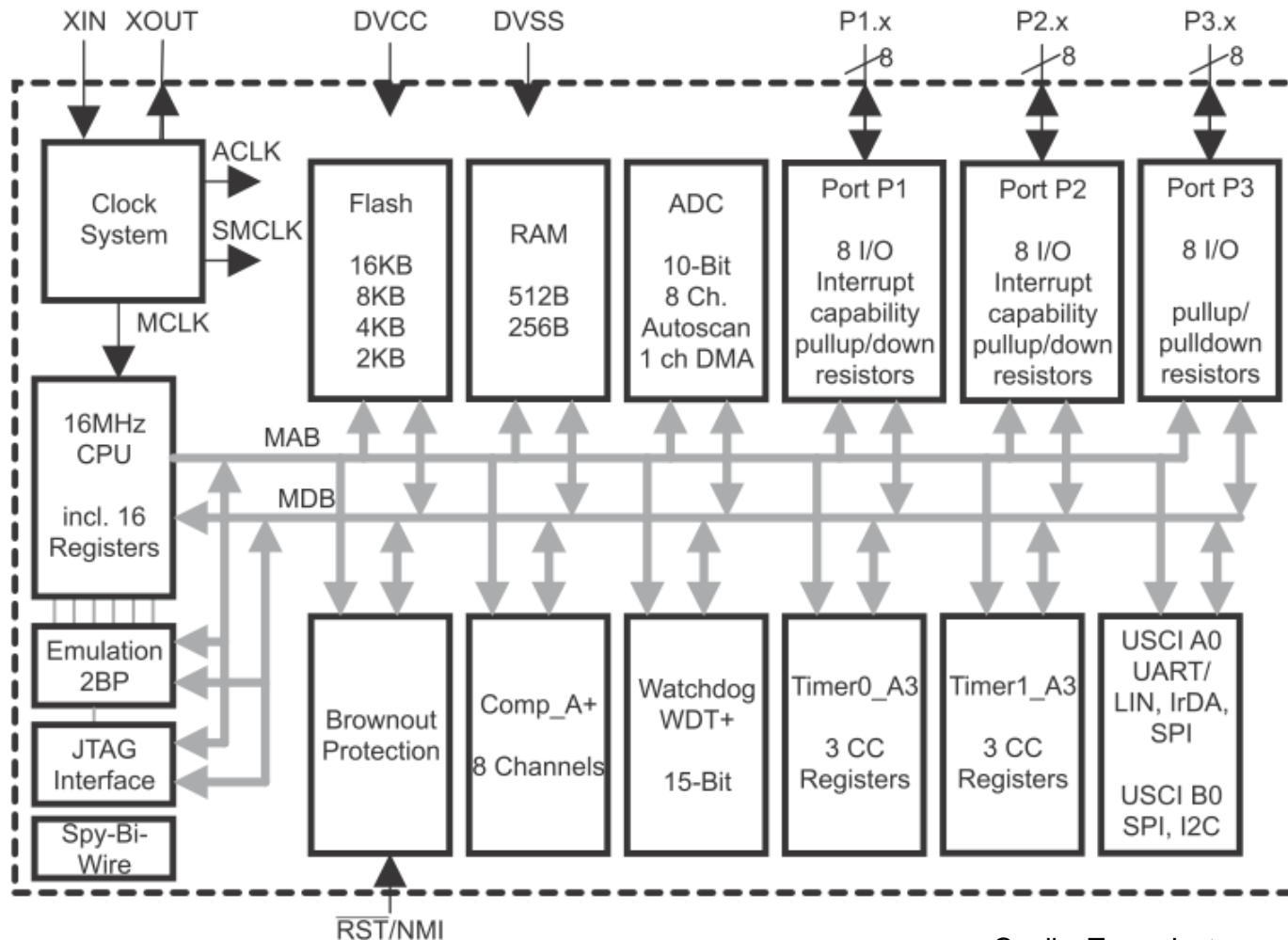
CPU: Central Processing Unit
RAM: Random Access Memory
ROM: Read-Only Memory

Beispiel:

Texas Instruments MSP430

4. Zeiger, Felder und
Zeichenketten

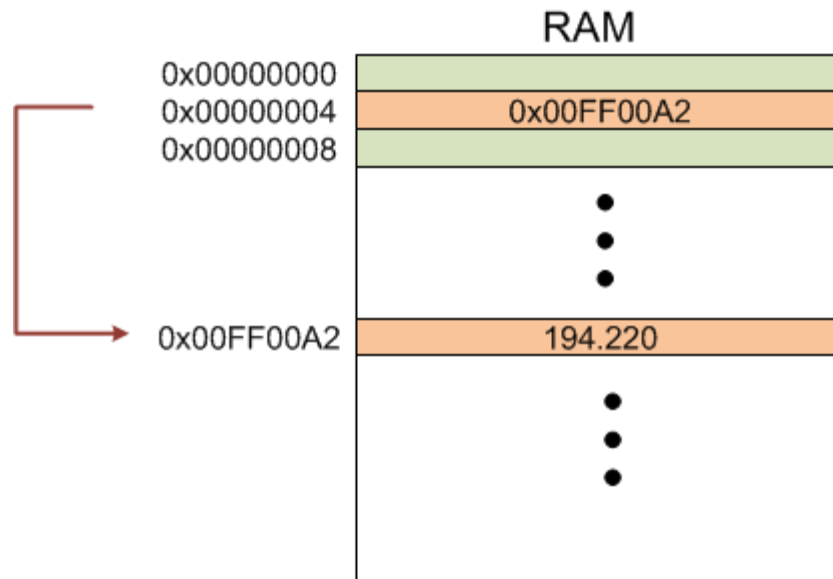
Functional Block Diagram, MSP430G2x53



Quelle: Texas Instruments SLAS735J

Zeiger (Pointer)

- Ein Zeiger ist eine Variable, die eine Speicheradresse (einer anderen Variable) enthält



- Zeiger werden wie Variablen definiert, allerdings mit vorangestelltem *
Syntax:

`<Datentyp> *ZeigerName;`

ZeigerName ist eine Zeigervariable, die auf eine Variable vom Typ
`<Datentyp>` zeigt

Zeiger (Pointer)

Operatoren

- Für die Ermittlung einer Speicheradresse (und Zuweisung an eine Zeigervariable) wird häufig der Adress-Operator **&** verwendet

Syntax:

```
ZeigerName = &Variable; // die Speicheradresse der Variable  
                        // wird der Zeigervariable zugewiesen
```

- Um auf den Speicher, auf den **ZeigerName** zeigt, zugreifen zu können, wird der Verweis-Operator ***** verwendet

Syntax:

```
*ZeigerName = 42;
```

- Beim Inkrementieren eines Zeigers wird die Adresse um **sizeof(<Datentyp >)** erhöht
- Void-Pointer (**void ***) sind Zeiger auf einen unbestimmten Datentyp

Zeiger (Pointer)

4. Zeiger, Felder und Zeichenketten

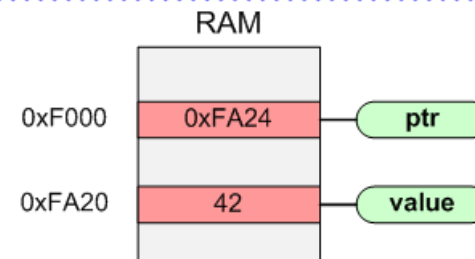
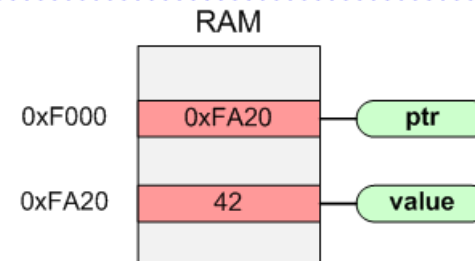
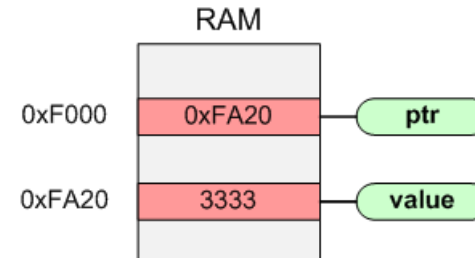
// Quelltext

```
int *ptr;  
int value=3333;
```

```
ptr = &value;
```

```
*ptr = 42;
```

```
ptr++;
```



Felder

- Felder sind aufeinanderfolgende Anordnungen von Elementen gleichen Typs
- Auf die einzelnen Elemente wird mit einem Index (positive Ganzzahl, **beginnend mit 0**) zugegriffen
- Felder werden im Speicher in aufsteigender Reihenfolge abgelegt
- Ein Feld ist durch die Anfangsadresse und die Anzahl und Größe der Elemente vollständig bestimmt
- Syntax (eindimensionales Feld \equiv Vektor):
<Datentyp> Name [Dimension] ;
- Mit der Definition kann auch eine Initialisierung verbunden sein, die Angabe der **Dimension** darf dann entfallen

- Die Indizierung beginnt mit der Zahl 0, bei einem Vektor mit N Elementen hat das letzte Element den Index $N-1$
- Beispiele:

```
int main(void)
{
    int    i;
    int    a[20];
    int    b[]    = {1,2,3,4};
    float  x[3]   = {0.1, 3.1};

    a[0] = b[0];
    a[1] = a[0] + 1;

    for(i=0;i<3;i++) {
        printf("x[%d] = %f\n", i, x[i]); /* Ausgabe:
                                           x[0] = 0.1
                                           x[1] = 3.1
                                           x[2] = 0.0 */
    }
}
```

- Mehrdimensionale Felder werden durch Hinzufügen weiterer Dimensionen definiert
- Beispiel:

```
int main(void)
{
    int i,j;
    int matrix[3][4] = {{11,12,13,14},
                        {21,22,23,24},
                        {31,32,33,34}};

    int tabelle[2][3][4];

    for(i=0;i<3;i++)
    {
        for(j=0;j<4;j++)
        {
            tabelle[1][i][j] = matrix[i][j];
            printf("%3d ",matrix[i][j]);
        }
        printf("\n");
    }
}
```

Zeiger und Felder

- Unterschiede und Gemeinsamkeiten -

- Die dynamische Anforderung von Speicher (zur Programmlaufzeit) erfolgt mit Hilfe von Bibliotheksfunktionen. Die Anforderung wird durch das Betriebssystem bearbeitet

- Zunächst muss ein Zeiger definiert werden:

Syntax:

```
<Datentyp> *Name ;
```

- Die Anforderung erfolgt durch Aufruf der Funktion `malloc()`

Syntax:

```
Name = (<Datentyp> *) malloc(Dimension*sizeof(<Datentyp>)) ;
```

- Falls kein ausreichender Speicherplatz zur Verfügung steht, gibt `malloc()` den Wert `NULL` zurück. Ansonsten liefert `malloc()` den Zeiger auf das erste Element.

- Der Speicher muss später wieder freigegeben werden:

Syntax:

```
free(Name) ;
```

- Siehe auch `realloc()`

Beispiele:

```
int *a;
int n;

n = 10;
a = (int *)malloc(n*sizeof(int));

a[0] = 1;

for(i=1; i<n; i++)
{
    a[i] = a[i-1]+2;
}

free(a);
```

```
int *a;
int b[10];

a = b;

a[0] = 1;

for(i=1; i<n; i++)
{
    *(a+i) = *(a+i-1)+2;
}
```

- Auf ein Feldelement kann mittels des Index zugegriffen werden, z.B.:

```
a = vector[0];
```

- Das Element mit dem Index 0 ist das erste Element des Feldes, dessen Adresse mit dem Adress-Operator & ermittelt werden kann, z.B.:

```
ptr = &vektor[0];
```

- Da der Vektor durch seine Anfangsadresse dargestellt wird, ist dies gleichbedeutend mit:

```
ptr = vektor;
```

- Der Zeiger ist selbst eine Variable, auf die arithmetische Operationen angewendet werden können:

```
int a, b;
```

```
a = *ptr;
```

```
b = *(ptr+3);
```

```
// entspricht: a = vektor[0];
```

```
// entspricht: b = vektor[3];
```

Felder und Zeichenketten

Felder, Vektoren, Arrays

Zeichenketten (Strings)

- Zeichenketten (Strings) sind Vektoren (eindimensionale Felder) aus Zeichen
- In C wird das Ende einer Zeichenkette mit einem `'\0'`-Zeichen markiert
- Fehlerquelle: Das abschließende `'\0'`-Zeichen benötigt Speicherplatz!
- Beispiele:

```
char s1[10] = "Hallo"; // Vektor mit 10 Elementen, die mit
                        // „Hallo“ initialisiert werden

char s2[] = "Hallo";   // Vektor mit 6 Elementen, die mit
                        // „Hallo“ initialisiert werden

char *s3 = "Hallo";    // Zeiger auf einen konstanten Vektor
                        // mit 6 Elementen, der „Hallo“ enthält
```

- Zur Zeichenkettenbearbeitung steht eine umfangreiche Bibliothek zur Verfügung (z.B. Kopieren von Strings mit `strncpy()`)

Nachtrag:

Zeichen- und String-Konstanten

- Zeichenkonstanten:
 - Einzelnes Zeichen, durch einfache Anführungszeichen eingeschlossen, z.B.: **'B'**
 - Zeichenkonstanten haben den Typ **int** und können wie ganzzahlige Konstanten benutzt werden (ASCII-Code)
 - Sonderzeichen werden durch Kombination mit einem Backslash dargestellt, z.B.: **'\n'** (= Zeilenvorschub)
- Stringkonstanten:
 - Zeichenfolge, die durch doppelte Anführungszeichen eingeschlossen wird, z.B.: **"Guten Tag"**
 - Die Zeichenfolge wird intern ohne Anführungszeichen und mit einem abschließenden **'\0'**-Zeichen gespeichert (bei Speicherallozierung berücksichtigen)
 - Für Sonderzeichen gelten die gleichen Regeln wie bei den Zeichenkonstanten
 - Im Quelltext kann ein String über mehrere Zeilen geschrieben werden, wenn unmittelbar vor dem Zeilenumbruch ein **'\'** eingefügt wird, z.B.:

```
"Dies ist eine sehr \
lange Zeichenkette"
```

Die Parameter der Funktion `main()`

Kommandozeilen-Parameter und main()

- Kommandozeilen-Parameter werden vom Betriebssystem beim Programmaufruf an die Funktion `main()` übergeben
- Kommandozeilenparameter werden durch Leerzeichen voneinander getrennt und stellen für das Betriebssystem jeweils separate **Zeichenketten** dar
- Syntax:

```
int main(int argc, char *argv[])
```

`argc` enthält die Anzahl der Kommandozeilen-Parameter

`argv[n]` ist ein Feld von Zeigern auf Zeichenketten. Die Zeichenketten enthalten die Kommandozeilenparameter

Anm: `argv[0]` zeigt auf den Programmnamen

- Eine Zeichenkette kann beispielsweise mit der Funktion `atoi()` in einen integer Wert bzw. mit `atof()` in einen double Wert konvertiert werden

Kommandozeilen-Parameter

Beispiel

Beispiel:

```
/* prog.c -> gcc -Wall prog.c -o prog*/

int main(int argc, char *argv[])
{
    int n;

    for(n=0; n<argc; n++)
    {
        printf("argv[%d]=\"%s\\\"\\n",n,argv[n]);
    }
}
```

```
$ prog
argv[0]="prog"

$ prog 4711 test
argv[0]="prog"
argv[1]="4711"
argv[2]="test"
```

Zeiger als Funktionsparameter

Zeiger und zusammengesetzte Datentypen

- Betrachte Struktur einer Funktionsdefinition:

```
<Typ> FunktionsName( <Parameterliste> )  
{  
    <Anweisungen>  
}
```

- <Typ>** gibt den Datentyp des Rückgabewertes an
Wird als Datentyp **void** angegeben, hat die Funktion keine Rückgabe
- <Parameterliste>** besteht aus Typ und Namen der an die Funktion übergebenen Parameter
- Werden Variablen(werte) der aufrufenden Funktion als Parameter übergeben, so wird innerhalb der aufgerufenen Funktion mit einer Kopie gearbeitet (**call by value**)
- Es kann auch die Adresse einer Variablen als Parameter übergeben werden. Damit kann innerhalb der aufgerufenen Funktion auf diese Variable zugegriffen werden (**call by reference**)

- Beim Funktionsaufruf werden die Argumente kopiert (Stack).
Innerhalb der Funktion verhalten sich die Parameter wie lokal definierte und durch die aufrufende Funktion initialisierte Variablen.
- Die Übergabe einer Adresse (call by reference) ermöglicht
 - der aufgerufenen Funktion Variablen der aufrufenden Funktion zu modifizieren
 - eine effiziente Parameterübergabe, da nur die Adresse, nicht aber der komplette Variableninhalt, kopiert werden muss

Zeiger und zusammengesetzte Datentypen

struct, union

4. Zeiger, Felder und Zeichenketten

Ausgehend von der Adresse einer Struktur-Variable oder einer Union-Variable kann mit dem ' \rightarrow '-Operator auf ein Strukturelement zugegriffen werden:

```
typedef struct
{
    int xpos;
    int ypos;
    int breite;
    int hoehe;
} Rechteck_typ;

long berechneFlaeche (Rechteck_typ r)
{
    return( r.breite * r.hoehe );
}

void verschiebeFlaeche (Rechteck_typ *r,
                        int dx,
                        int dy)
{
    r->xpos += dx;
    r->ypos += dy;
}

int main(void)
{
    long          flaeche;
    Rechteck_typ fenster = {30,10,100,200};

    verschiebeFlaeche (&fenster,20,-5);
    flaeche = berechneFlaeche(fenster);
}
```


Zeiger und zusammengesetzte Datentypen

Beispiel 2

4. Zeiger, Felder und Zeichenketten

```
typedef enum {
    NONE = 0,
    RECHTECK,
    KREIS
} Form_enum;

typedef struct {
    float breite;
    float hoehe;
} Rechteck_typ;

typedef struct {
    float radius;
} Kreis_typ;

typedef struct {
    Form_enum form;
    float      xpos;
    float      ypos;
    union
    {
        Rechteck_typ recht;
        Kreis_typ     kreis;
    };
} Flaeche_typ;
```

```
float berechneFlaeche(Flaeche_typ *f)
{
    switch(f->form) {
        case RECHTECK:
            return(f->recht.breite*f->recht.hoehe);
            break;
        case KREIS:
            return(M_PI*f->kreis.radius*f->kreis.radius);
            break;
        default:
            return( -1 );
            break;
    }
}

int main(void)
{
    int      i=0;
    float     finhalt;
    Flaeche_typ f[]={RECHTECK,1,2,{{5,10}}},
                {KREIS      ,3,5,{{2}}},
                {NONE}};

    while((finhalt = berechneFlaeche(&f[i++])) >= 0)
    {
        printf("Flaeche [%d] = %f\n",i,finhalt);
    }
}
```

Zeiger auf Funktionen

- Zeiger auf Funktionen werden sehr häufig im Kontext von Schnittstellen (Plugin, Betriebssystem-Module usw.) eingesetzt

- Syntax:

```
<Datentyp> (*FunktionsZeiger) (<Parameterliste>);
```

FunktionsZeiger ist ein Zeiger auf eine Funktion, die einen Rückgabewert vom Typ **<Datentyp>** hat.

Die **<Parameterliste>** kann auch leer sein.

- Zur Initialisierung des Funktionsname wird der Variablen eine Funktion zugewiesen:

```
FunktionsZeiger = Name_einer_Funktion;
```

- Um die Funktion aufzurufen (De-Referenzierung), kann jetzt der Zeiger verwendet werden:

```
(*FunktionsZeiger) (<Parameter>);
```

oder

```
FunktionsZeiger (<Parameter>);
```

```
/* objekte.c */

typedef struct
{
    int hoehe, breite, radius;
    void (*calcFlaeche) ();
} Form_t;

void Flaeche(Form_t *form)
{
    form->calcFlaeche(form);
}

void calcViereck(Form_t *form)
{
    printf("Viereck: %f\n",
           (float)form->hoehe
           *(float)form->breite);
}

void calcKreis(Form_t *form)
{
    printf("Kreis: %f\n",
           M_PI*pow(form->radius,2));
}
```

```
void createKreis(Form_t *form, int radius)
{
    form->radius      = radius;
    form->calcFlaeche = calcKreis;
}

void createViereck(Form_t *form, int
                                                             int
    hoehe,
    breite)
{
    form->hoehe      = hoehe;
    form->breite      = breite;
    form->calcFlaeche = calcViereck;
}

int main(void)
{
    int i;
    Form_t form[3];

    createKreis (&form[0],2);
    createViereck(&form[1],5,4);
    createViereck(&form[2],1,2);

    for(i=0;i<3;i++)
        Flaeche(&form[i]);
}
```

Zeiger auf Funktionen

Beispiel 2

4. Zeiger, Felder und
Zeichenketten

```
/* state.c */

void state_A(void);
void state_B(void);
void state_C(int flag);

void (*stateFkt) () = state_A;

int main(void)
{
    int durchlauf=0;
    while(1)
    {
        printf("Zustand = ");
        stateFkt(durchlauf++);
    }
}
```

```
void state_A(void)
{
    printf("A:");
    switch(getchar())
    {
        case 'b': stateFkt = state_B; break;
        case 'c': stateFkt = state_C; break;
        case 'e': stateFkt = exit;      break;
    }
    while(getchar() != '\n');
}

void state_B(void)
{
    static int cnt=0;

    printf("B:cnt=%d\n",cnt);
    if(cnt == 3)
        stateFkt = state_C;
    cnt = (cnt == 3)?0:cnt+1;
}

void state_C(int flag)
{
    printf("C:%d\n",flag);
    stateFkt = state_A;
}
```