2 Sprachkonzepte von C

Motivation:

In den folgenden Abschnitten werden die charakteristischen Operatoren, Datenstrukturen und Steueranweisungen sowie das Zeigerkonzept von C vorgestellt.

2.1 Spezielle Operatoren

Motivation:

In diesem Abschnitt werden die für die Sprache C typischen Operatoren vorgestellt, die in anderen höheren Programmiersprachen nicht oder in anderer Form vorhanden sind.

2.1.1 Inkrement-und Dekrementoperatoren

Es gibt die unären Operatoren ++ zur Inkrementierung und -- zur Dekrementierung des Operanden um den Wert 1. Beide Operatoren können nur auf einen Operanden angewendet werden, der sich auf einen modifizierbaren Speicherbereich (lvalue) bezieht.

allg.: ++ lvalue /* Präfixnotation */
-- lvalue
lvalue ++ /* Postfixnotation */
lvalue --

- Präfixnotation: Inkrementierung und Dekrementierung **vor** Wertberechnung des Ausdruckes
- Postfixnotation: Inkrementierung und Dekrementierung **nach** Wertberechnung des Ausdruckes

```
int x, y;
x = 3;
y = ++x;    /* Praefixinkrementierung: x = x + 1; y = x; */
x = 3;
y = x++;    /* Postfixinkrementierung: y = x; x = x + 1; */
x = 3;
++x;        /* ist identisch zu: x++; */
```

2.1.1 Inkrement-und Dekrementoperatoren

```c
#include <stdio.h>

main()          /* Inkrement-Operatoren */
{       int x , y, z;

        x = y = 3;
        z = x++ - 2;
        printf("x:%d y:%d z:%d\n",x,y,z);
        z = -x++ + ++y;
        printf("x:%d y:%d z:%d\n",x,y,z);
        z = x / ++x; /* ++ hat Vorrang vor / */
        printf("x:%d y:%d z:%d\n",x,y,z);
}
```

```c
#include <stdio.h>

long fakult(n)                          /* Berechnung von n! */
int n;
{
   long i, fak;

   fak = i = 1;
   while (++i <= n)
        fak = fak * i;
   return(fak); /* Ergebnis an main-Funktion uebergeben */
}

main()                  /* Argument ist Ergebnis von fakult */
{
  printf("10! = %d\n", fakult(10));
}
```

2.1.2 Bitorientierte Operatoren

Bitorientierte Operatoren ermöglichen die direkte Manipulation einzelner Bits:

~	Einerkomplement	~x
&	bitweise Konjunktion (UND)	x & 036
^	bitweise Antivalenz (exklusives ODER)	x ^ 0100
|	bitweise Disjunktion (inklusives ODER)	x | y
>>	bitweise Rechtsverschiebung	x >> 2
<<	bitweise Linksverschiebung	y << 1

2.1.2 Bitorientierte Operatoren

Die Operanden logischer Operatoren werden bitweise verknüpft nach folgender Regel:

x	y	x & y	x | y	x ^ y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

2.1.2 Bitorientierte Operatoren

Bedingungen für Operanden:

- müssen von integralem Typ sein
- Verschiebewert positiv und kleiner als Wortlänge
- Auffüllen mit 0 oder 1 (bei Rechtsverschiebung negativer Zahlen)

```
//x :                          0101 0001
   x << 4;            //==>    0001 0000
   x >> 4;            //==>    0000 0101
   ~ x;               //==>    1010 1110

//y:                           0011 1001
   x & y;             //==>    0001 0001
   x | y;             //==>    0111 1001
   x ^ y;             //==>    0110 1000
```

2.1.2 Bitorientierte Operatoren

```
#include <stdio.h>
int wordlen()
{
   int i; unsigned wort;

   i = wort = 0;
   wort = ~wort;                          /* alle Bits 1 */
   while (wort != 0) {
        wort = wort >> 1;
        i++;
   }
   return(i);
}

main()  /* Zaehlen der Bits eines Maschinenewortes */
{
   printf("Anzahl der Bits: %d\n",wordlen());
}
```

2.1.3 Zusammengesetzte Zuweisungsoperatoren

Zusammengesetzte Zuweisungsoperatoren sind eine verkürzte Schreibweise für Wertzuweisungen der allg. Form:

lvalue operator= ausdruck;

```
a += b;        // a = a + b
x *= y - 2 ;   // x = x * (y - 2)
i >>= 1;       // i = i >> 1
```

binäre arithmetische Operatoren: += -= *= /= %=

binäre bitorientierte Operatoren: &= |= ^= <<= >>=

```
#include <stdio.h>

main() /* (zusammengesetzte) Zuweisungsoperatoren */
{
        int x = 4, y, z = 1;

        printf("%d\n",    x *= z + 4        );
        printf("%d\n",    y = x -= z * 8    );
        printf("%d\n",    z = y == x        );
        printf("%d\n",    y == ( x = z )    );
}
```

2.1.4 Entscheidungsoperator

Der Entscheidungsoperator ? : verknüpft 3 Ausdrücke

```
ausdruck1 ? ausdruck2 : ausdruck3
```

miteinander und ist die verkürzte Schreibweise einer Zweiwegeentscheidung.

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

Steht für

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

Der Entscheidungsoperator erzeugt einen Ausdruck, der selbst wieder in Ausdrücken als Operand genutzt werden kann. Sinnvolle Anwendung z.B. beim Funktionsaufruf in Parameterlisten.

2.1.5 Kommaoperator

Der Kommaoperator fasst mehrere Ausdrücke zu einem zusammen. Er wird benutzt, wenn man mehrere Ausdrücke an einer Stelle unterbringen muss, wo eigentlich nur ein Ausdruck erlaubt ist.

```
allg.: ausdruck1 , ausdruck2     /* ist wieder ein Ausdruck */
```

```
if (x = 1, x == y)
```

Als Wert des Gesamtausdrucks steht der Wert des letzten Teilausdrucks. Wird in der Regel nur in for-Schleifen angewendet.

```
for(  x = 0, y = 5; x < y ; x++ , y--)
```

2.1.6 Vorrangregeln und Assoziativität

Operatoren	Bezeichnung	Assoziativität
( ) [ ] -> .	Funktions- / Ausdrucksklammer Feldindizierung Zeiger auf Strukturelement Strukturelementvariable	L => R
! ~ ++ -- - (*typ*) * & sizeof	Negation Einerkomplement Inkrement Dekrement Vorzeichen Minus explizite Typkonvertierung Zugriff über Zeiger Adresse des Datenobjektes Größe des Datenobjektes in Bytes	R => L
* / %	Multiplikation Division Modulo	L => R
+ -	Addition Subtraktion	L => R
<< >>	Linksverschiebung von Bits Rechtsverschiebung von Bits	L => R
< <= > >=	Test kleiner Test kleiner gleich Test größer Test größer gleich	L => R

Operatoren	Bezeichnung	Assoziativität
== !=	Test gleich Test ungleich	L => R
&	bitweises UND	L => R
^	bitweises exklusives ODER	L => R
|	bitweises inklusives ODER	L => R
&&	logisches UND	L => R
||	logisches ODER	L => R
?:	bedingte Zuweisung	R => L
= *op*=	Wertzuweisungen *op*: + - * / % & | ^ <<>>	R => L
,	Kommaoperator	L => R

2.2.2 Übliche arithmetische Typumwandlung

Die Operanden eines binären Operators dürfen in C einen unterschiedlichen skalaren Datentyp besitzen. Durch die übliche arithmetische Datentypumwandlung wird dabei implizit ein gemeinsamer Datentyp gebildet. Dazu wird die ganzzahlige Erweiterung ausgeführt.

- Bei ganzzahligen Standardtypen gilt folgende Rangfolge:
  - `char < short < int < long < long long`

- Die Rangfolge der Gleitpunkttypen sieht folgendermaßen aus:
  - `float < double < long double`

2.2.2 Übliche arithmetische Typumwandlung

Die automatische Typumwandlung (implizit) funktioniert nicht bei den Zuweisungsoperatoren und den logischen Operatoren && und ||.

Typ	Operand	Typ	Umwandlung
`int`	`=`	`float`	Der Nachkommateil wird weggelassen.
`int`	`=`	`double`	
`int`	`=`	`long`	Die höherwertigen Bits werden weggelassen.
`char`	`=`	`int`	
`char`	`=`	`short`	
`float`	`=`	`double`	Der Wert wird entweder gerundet oder abgeschnitten (hängt von der Implementierung ab).
`float`	`=`	`long, int, short, char`	Sollte hier keine genaue Darstellung möglich sein, wird der Wert entweder gerundet oder abgeschnitten (ebenfalls abhängig von der Implementierung).
`double`	`=`		

2.3 Funktionen

Motivation:

Ein C-Programm besteht aus einer Menge (nicht geschachtelter) Funktionen. Sie können über mehrere separate Quellfiles verteilt werden. Die Programmausführung beginnt immer mit der Funktion main.

2.3 Funktionen

Funktionen sind kleine Unterprogramme, mit denen Sie Teilprobleme einer größeren Aufgabe lösen können.

In der Praxis können Sie sich das so vorstellen:

Eine Funktion führt eine komplizierte Berechnung aus, eine andere Funktion schreibt das Ergebnis der Berechnung in eine Datei, und wieder eine andere überprüft das Ergebnis auf Fehler.
Die parallele Ausführung von Funktionen ist in ANSI C allerdings nicht möglich. Funktionen werden also in der Regel wie normale Anweisungen auch nur sequenziell, das heißt nacheinander, ausgeführt.

2.3.1 Vereinbarungen von Funktionen

Die Definition einer Funktion besteht aus einem Funktionskopf und einem Funktionskörper.

Kernighan:

```
speicherklasse typ name ( params)

parameter_deklarations_liste
{
    vereinbarungs_liste
    anweisungs_liste
}
```

ANSI-C:

```
speicherklasse typ name ( params_deklarations)
{
    vereinbarungs_liste
    anweisungs_liste
}
```

2.3.1 Vereinbarungen von Funktionen

typ: Hier legen Sie den Datentyp des Rückgabewerts fest. Dabei dürfen Sie alle Datentypen verwenden Eine Funktion ohne Rückgabewert wird als `void` deklariert. Sollten Sie einmal keinen Rückgabetyp angeben.

Name: Dies ist ein eindeutiger Funktionsname, mit dem Sie die Funktion von einer anderen Stelle aus im Programmcode aufrufen können. Für den Funktionsnamen selbst gelten dieselben Regeln wie für Variablen.

params_deklarations:
Die Parameter einer Funktion sind optional. Sie werden durch einen Datentyp und einen Namen spezifiziert und durch ein Komma getrennt. Wird kein Parameter verwendet, können Sie zwischen die Klammern entweder void schreiben oder gar nichts.

speicherklasse:
Außerdem lassen sich bei Funktionen auch sogenannte Speicherklassen-Spezifizierer verwenden. Mehr hierzu finden Sie im entsprechenden Abschnitt.

2.3.2 Funktionsaufruf

```
void hilfe(void)
{
   printf("Ich bin die Hilfsfunktion\n");
}
```

Die Funktion hat keinen Rückgabetyp und keine(n) Parameter. Aufgerufen wird sie mit :

```
   hilfe();
```

innerhalb der main()-Funktion. Sehen Sie sich den Quellcode dazu an:

```
void hilfe(void)
{
   printf("Ich bin die Hilfsfunktion\n");
}

int main(void)
{
   hilfe();
}
```

2.3.3 Funktionsdeklaration

Damit der Compiler überhaupt von einer Funktion Kenntnis nimmt, muss diese vor ihrem Aufruf deklariert werden. Im vorangegangenen Beispiel ist das automatisch geschehen. Zuerst wurde die Funktion `hilfe()` und anschließend die `main()`-Funktion definiert.

```
void hilfe(void);

int main(void)
{
    hilfe();
    return 0;
}

void hilfe(void)
{
    printf("Ich bin die Hilfsfunktion\n");
}
```

2.3.3 Funktionsdeklaration

Da hier die Funktion `hilfe()` erst hinter der `main()`-Funktion geschrieben wurde, müssen Sie den Compiler zur Übersetzungszeit mit dieser Funktion bekannt machen. Sonst kann der Compiler in der `main()`-Funktion mit `hilfe()` nichts anfangen. Dies stellen Sie mit einer sogenannten Vorwärtsdeklaration sicher:

```
void hilfe(void);
```

Die Deklaration wird im Gegensatz zur Funktionsdefinition mit einem Semikolon abgeschlossen. Funktionsdeklarationen sollten aber nicht nur dann vorgenommen werden, wenn die Funktionen hinter die main()-Funktion geschrieben werden. Es ist ja auch möglich, dass eine Funktion andere Funktionen aufruft

2.3.4 Rekursive Funktionen

Eine Rekursion ist eine Funktion, die sich selbst aufruft und sich selbst immer wieder neu definiert. Damit sich aber eine Rekursion nicht unendlich oft selbst aufruft, sondern irgendwann auch zu einem Ergebnis kommt, benötigen Sie unbedingt eine sogenannte Abbruchbedingung. Sonst kann es irgendwann passieren, dass Ihr Computer mit einem Stacküberlauf oder Stack-Overflow abstürzt.

```
}
```

Rekursive Funktionen

Eine Funktion soll zwei Zahlen dividieren. Der ganzzahlige Rest der Division soll angegeben werden. Zum Beispiel: 10/2=5 oder 10/3=3 Rest 1. Das Programm darf aber nicht die Operatoren / und % verwenden. Die Lösung soll die Form einer rekursiven Funktion haben:

$$divide(x,y) = \begin{cases} x \geq y : 1 + divide(x-y\,,y) \\ x < y : 0 \end{cases}$$

```
int divide(int x, int y)
{
     if(x >= y)
          return (1 + divide(x - y, y));
     if(x)
          printf("Zahl nicht teilbar -> Rest: %d -> ", x);

     return 0;

}
```

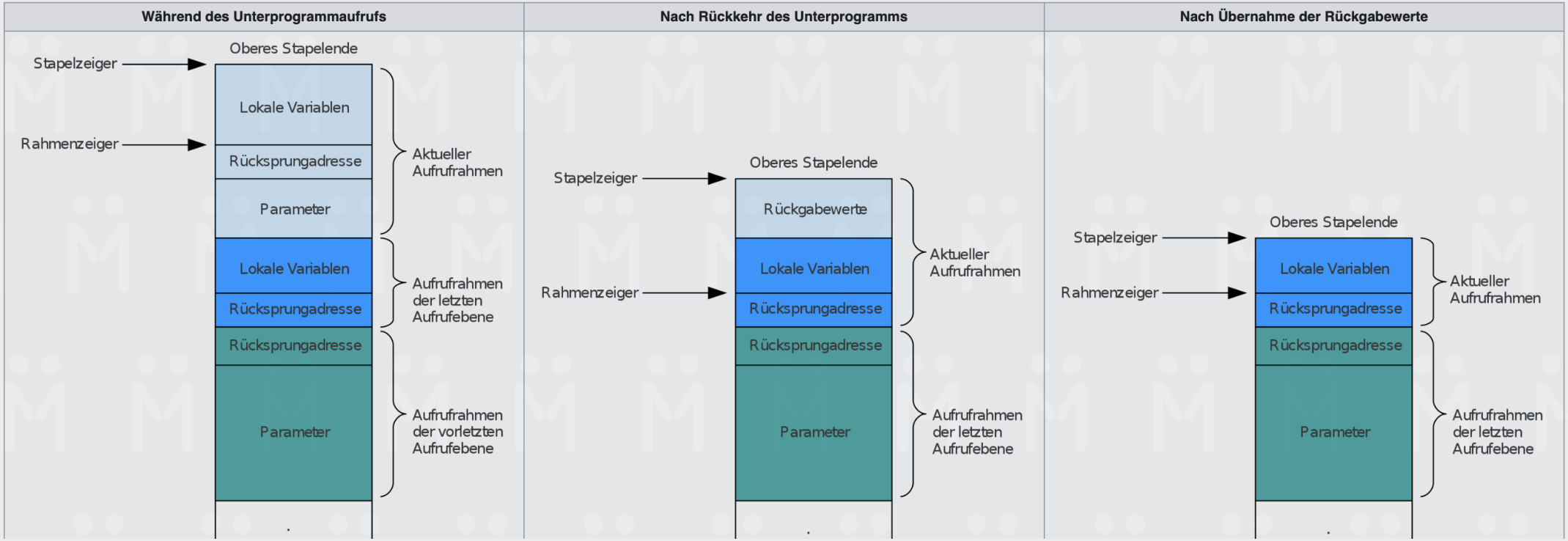
2.3.4 Stack

Wenn eine Funktion aufgerufen wird, erweitert der Compiler den Stack um einen Datenblock. In diesem Datenblock werden die Parameter, die lokalen Variablen und die Rücksprungadresse zur aufrufenden Funktion angelegt. Dieser Datenblock wird als Stack-Frame oder Stackrahmen bezeichnet.

Der Datenblock bleibt so lange bestehen, bis diese Funktion wieder endet. Wird in ihm aber eine weitere Funktion aufgerufen, wird ein weiterer Datenblock auf den (richtig wäre: unter den) aktuellen gepackt. Der Stack wächst nach unten an. Am Anfang des Stacks befindet sich der Startup-Code, der die main()-Funktion aufruft, die eine Position unter dem Startup-Code liegt. An unterster Stelle befindet sich immer die aktuelle Funktion, die gerade ausgeführt wird. Eine Position – oder besser: einen Datenblock – darüber liegt die aufrufende Funktion in der Wartestellung. Sie wartet auf die Beendigung der nächsten aufgerufenen Funktion.

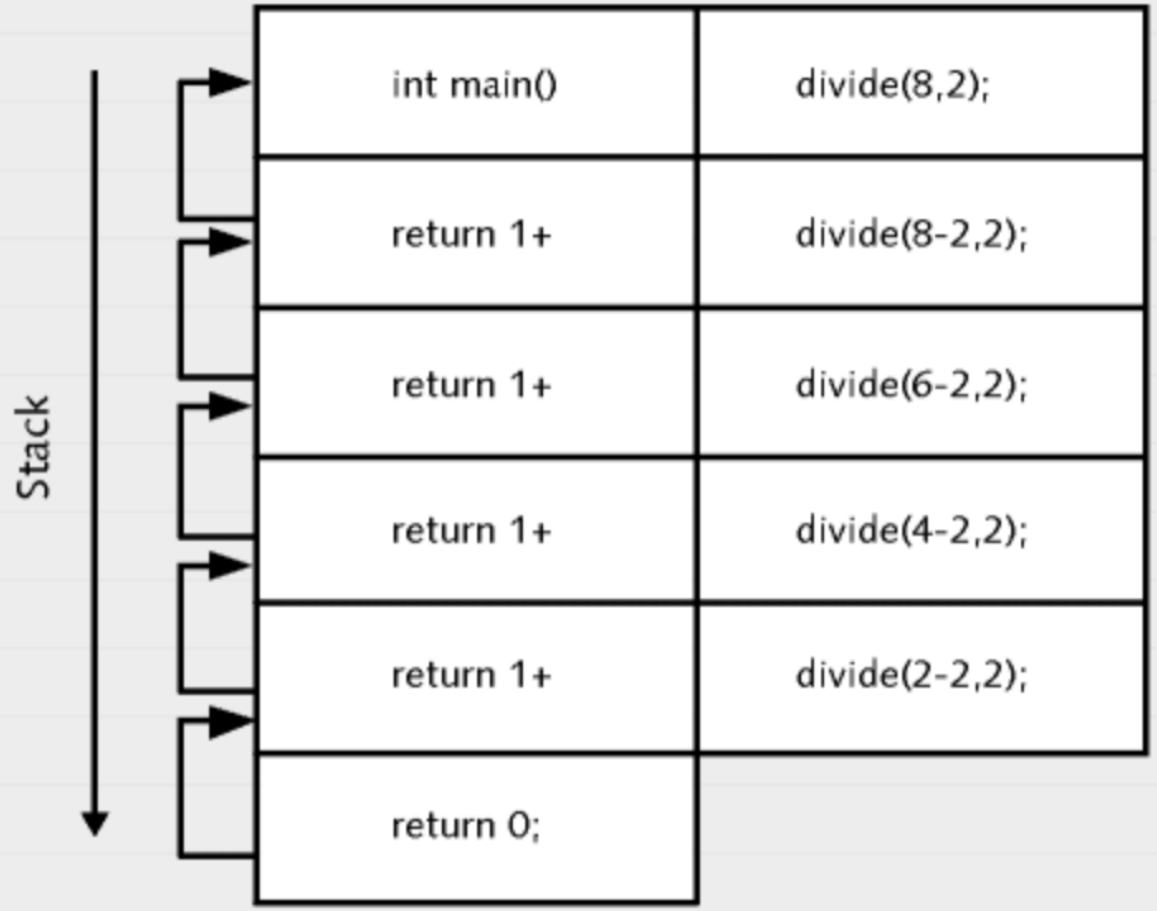
2.3.4 Call-Stack (Intel x86)

```
double f3(double p)
{return p * 2.0;}

double f2(int p)
{return  f3( p: 2.0) * p;}

void f1()
{int x = f2( p: 2);}
```



2.3.4 Stack

```
/* Funktionsaufruf */
printf("8/2 = Ergebnis : %d\n", divide(8, 2));
```



2.3.5 Globale Variablen

Globale Variablen können Sie sich als Vorwärtsdeklarationen von Funktionen vorstellen. Und wie der Name schon sagt, sind globale Variablen für alle Funktionen gültig.

```
int i=333; /* globale Variable */

void aendern(void)
{
    i = 111;
    printf("In der Funktion aendern: %d\n",i); /* 111 */
}

int main(void)
{
    printf("%d\n",i);  /* 333 */
    aendern();
    printf("%d\n",i);  /* 111 */
    return 0;
}
```

2.3.6 Statische Variablen

Statische Variablen verlieren bei Beendigung ihres Bezugsrahmens (also bei Beendigung der Funktion) nicht ihren Wert, sondern behalten diesen bei.

Dass dies gelingt, liegt daran, dass statische Variablen nicht im Stacksegment der CPU, sondern im Datensegment gespeichert werden.

Achtung: **Statische Variablen müssen schon bei ihrer Deklaration initialisiert werden!**

2.3.6 Statische Variablen

```
void inkrement(void)
{
     static int i1 = 1;
     int i2 = 1;
     printf("Wert von i1: %d\n",i1);
     printf("Wert von i2: %d\n",i2);
     i1++;i2++;
}

int main(void)
{
     inkrement();    //i1 = 1 , i2 = 1
     inkrement();    //i1 = 2 , i2 = 1
     inkrement();    //i1 = 3 , i2 = 1
     return 0;
}
```

2.4 Arrays

Motivation:

Mit Arrays haben Sie die Möglichkeit, eine geordnete Folge von Werten eines bestimmten Typs abzuspeichern und zu bearbeiten. Arrays werden auch als Vektoren, Felder oder Reihungen bezeichnet.

2.4.1 Felder (Arrays)

Feld ist eine Zusammenfassung von Datenobjekten gleichen Typs
auf die Elemente des Feldes kann man mittels Indizierung zugreifen
das erste Feldelement hat immer den Index 0.

allg. Form einer Feldvereinbarung:

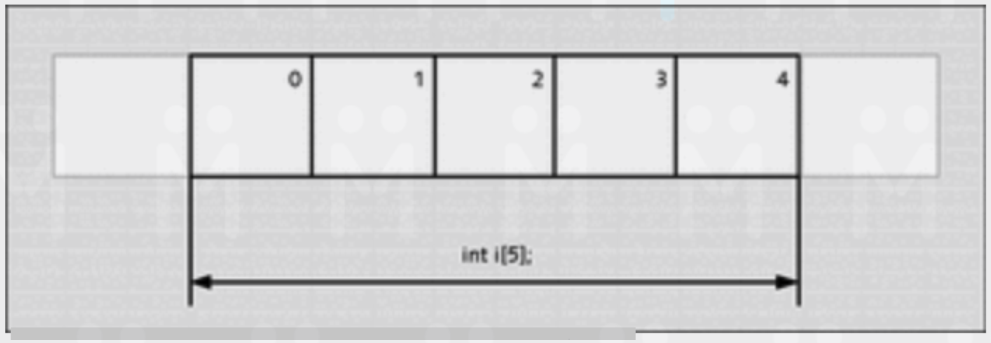
- `speicherklasse typ bezeichner [konst_ausdruck]`
- `konst_ausdruck muss einen Integerwert ergeben.`

```
#define N 1000
float farray[10];    /* 10 Elemente vom Typ float */
char  string[100];   /* Zeichenfeld für 100 Zeichen */
long  z[N];          /* 1000 long-Elemente */
```

2.4.1 Felder (Arrays)

```
int i[5];
```

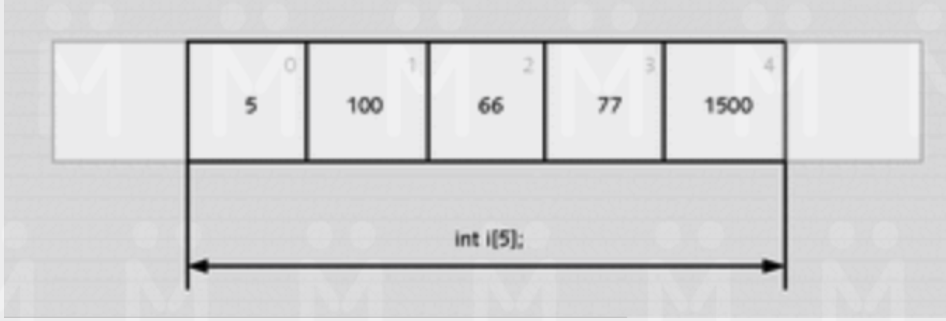
Durch diese Deklaration wird Platz für fünf Variablen im Array i vom Datentyp int im Speicher reserviert.



Mit dieser Deklaration wurde automatisch auch Speicherplatz für fünf int-Werte reserviert. Bei vier Bytes für eine int-Variable (je nach System) würden 20 Bytes im Arbeitsspeicher des Rechners belegt werden.

2.4.2 initialisieren von Arrays

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

int main(void) {
   int i[5];      /* Array mit 5 int-Elementen */

   /* Wertzuweisungen des Arrays */
   i[0] = 5;
   i[1] = 100;
   i[2] = 66;
   i[3] = 77;
   i[4] = 1500;

   /*Ausgabe der einzelnen Array-Elemente*/
   printf("Array-Element i[0]= %d\n", i[0]);
   printf("Array-Element i[1]= %d\n", i[1]);
   printf("Array-Element i[2]= %d\n", i[2]);
   printf("Array-Element i[3]= %d\n", i[3]);
   printf("Array-Element i[4]= %d\n", i[4]);
   return EXIT_SUCCESS;
}
```



2.4.2 initialisieren von Arrays

Arrays lassen sich auch anders, nämlich direkt bei der Deklaration, initialisieren. Die Werte müssen dabei zwischen geschweiften Klammern stehen:

```
int numbers[] = { 1, 2, 4, 5, 9 };
```

Wenn Sie das Array so initialisieren, können Sie die Größe des Arrays auch weglassen. C kümmert sich darum, dass genügend Speicher zur Verfügung steht. Die einzelnen Initializer werden immer mit einem Komma getrennt und stehen in geschweiften Klammern. Dadurch ist das Feld wie folgt mit Werten belegt:

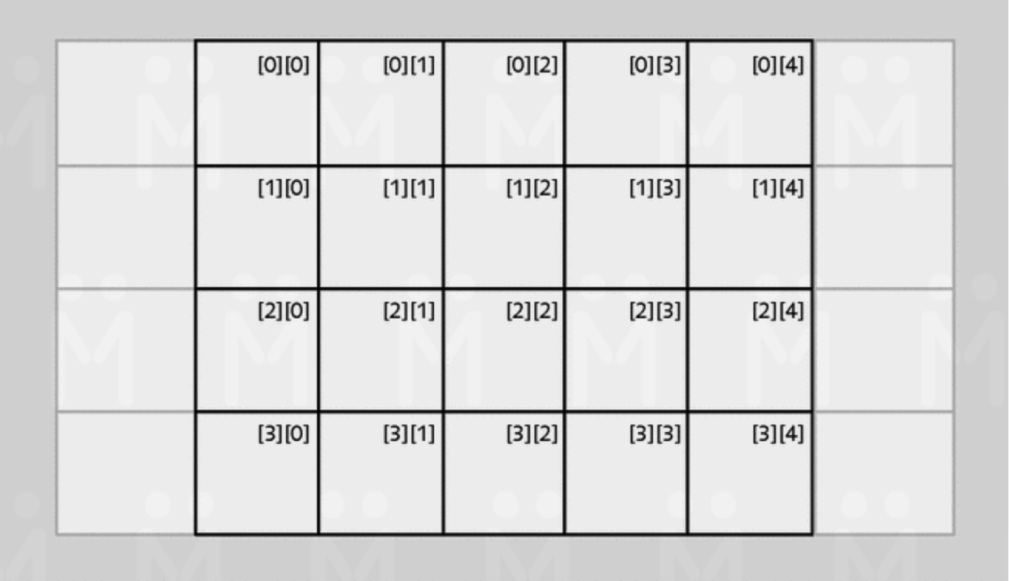
```
numbers[0] = 1;
numbers[1] = 2;
numbers[2] = 4;
numbers[3] = 5;
numbers[4] = 9;
```

2.4.3 Mehrdimensionale Arrays

Arrays sind ein Strang von hintereinander aufgereihten Zahlen. Man spricht dann von eindimensionalen Arrays oder Feldern. Es ist aber auch möglich, Arrays mit mehr als nur einer Dimension zu verwenden:

```
int Matrix[4][5]; /* Zweidimensional 4 Zeilen x 5 Spalten */
```

Hier wurde ein zweidimensionales Array mit dem Namen Matrix definiert. Dies entspricht im Prinzip einem Array, dessen Elemente wieder Arrays sind.



2.4.4 Mehrdimensionale Arrays initialisieren

```
/* 4 Zeilen 5 Spalten */

int Matrix[4][5] = { {10,20,30,40,50},
                     {15,25,35,45,55},
                     {20,30,40,50,60},
                     {25,35,45,55,65}};
```

	[0][0] 10	[0][1] 20	[0][2] 30	[0][3] 40	[0][4] 50	
	[1][0] 15	[1][1] 25	[1][2] 35	[1][3] 45	[1][4] 55	
	[2][0] 20	[2][1] 30	[2][2] 40	[2][3] 50	[2][4] 60	
	[3][0] 25	[3][1] 35	[3][2] 45	[3][3] 55	[3][4] 65	

2.4.5 Strings/Zeichenketten (»char«-Array)

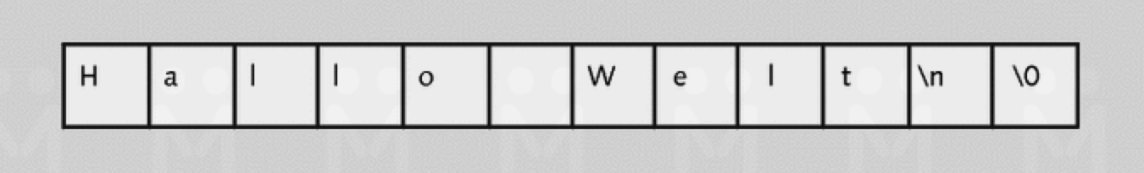
Arrays vom Datentyp `char` werden `Strings` genannt. Ein String ist eine Kette von einzelnen `char-Zeichen` mit einer abschließenden 0 (was nicht mit dem Zeichen '0' gleichzusetzen ist). char-Arrays sind typischerweise eindimensional.

Die Deklaration eines char-Arrays ist identisch mit der bisher bekannten Form der Array-Deklaration:

```
char string_array[100];
const char hallo[] = {'H','a`,'l','l','o',' ,,
                      'W','e','l','t','\n','\0`};
```

Diese Schreibweise ist sehr umständlich. Daher können Sie ein char-Array auch anders, nämlich als einen String, deklarieren:

```
const char hallo[] = { "Hallo Welt\n" };
```



2.4.5 Zeichneketten Funktionen

- strcat() – Strings aneinanderhängen

```
char *strcat(char *s1, const char *s2);
```

- strcmp() – Strings vergleichen

```
int strcmp(const char *s1, const char *s2);
```

Sind beide Strings identisch, gibt diese Funktion 0 zurück. Ist der String s1 kleiner als s2, ist der Rückgabewert kleiner als 0; und ist s1 größer als s2, dann ist der Rückgabewert größer als 0.

- strcpy() – einen String kopieren

```
char *strcpy(char *s1, const char *s2);
```

Dass hierbei der String-Vektor s1 groß genug sein muss, versteht sich von selbst. Bitte beachten Sie dabei, dass das Ende-Zeichen '\0' auch Platz in s1 benötigt

- strlen() – Länge eines Strings ermitteln

```
size_t strlen(const char *s1);
```

Die Länge des adressierten Strings s1 ohne das abschließende Stringende-Zeichen zurückgegeben

2.4.6 Übergabe von Arrays an Funktionen

Um Arrays an Funktionen zu übergeben, gehen Sie ähnlich wie bei Variablen vor.

Die Funktionsdeklaration sieht folgendermaßen aus:

```
void function(int feld[], int n_Anzahl)
```

Auffällig ist hier, dass der Indexwert für die Größe des Arrays nicht angegeben wird. Das liegt daran, dass der Funktion nicht bekannt ist, wie viele Elemente das Array besitzt.

Daher ist es empfehlenswert, der Funktion die Anzahl der Elemente als Argument mitzugeben, wie auch im Beispiel oben bei der Deklaration der Variablen `n_Anzahl` zu sehen ist.