Pointer – die Seele von c

- Pointer sind Adressen von Daten.
- Da Daten immer von einem bestimmten Typ sind, sind auch Pointer an einen Datentyp gebunden.
- Pointervariablen sind Variable, die als Wert die Adresse einer anderen Variablen enthalten.

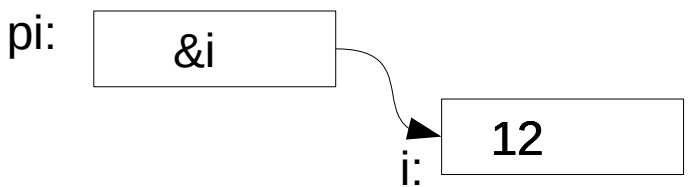
Pointer

- Die Adresse einer Variablen wird mit einem vorangestellten & gebildet.
- Ein * vor einem Pointer ist der Dereferenzierungsoperator. Er liefert den Wert, auf den ein Pointer zeigt.
- Pointer werden in printf mit %p ausgegeben.

```c
#include <stdio.h>

int main()
{
    int i=12;
    int*pi=&i;
    printf("i: %d, &i: %p, p-> %d\n",i,pi,*pi);
    return 0;
}
```

Pointervariable pi, Initialisiert mit der Adresse von i.

Dereferenzierter Pointer Liefert den Wert, auf den Die Adresse im Pointer zeigt.



- Der Pointer pi zeigt/verweist auf die Variable i.
- Der Wert der Pointervariablen ist die Adresse von i.
- Wird der Pointer dereferenziert, so erhält man den Wert, auf den der Pointer zeigt, hier der Wert von i, nämlich 12.

```
$ ./a.out
i: 12, &i: 0x7ffe00a9b8d8, p-> 12
```

Wenn Sie das Beispiel abschreiben und ausprobieren – und das sollten Sie mit allen Beispielen machen - wird der Adresswert ein anderer sein. Er kann sich bei jedem Aufruf des Programms ändern!

Pointer

```c
#include <stdio.h>

int main()
{
    int i=12;
    int j=21;
    int*pi=&i;
    printf("i: %d, &i: %p, p-> %d\n",i,pi,*pi);
    pi = &j;
    printf("j: %d, &i: %p, p-> %d\n",j,pi,*pi);
    return 0;
}
```

pi zeigt nun auf
Die Variable j
mit dem Wert 21.
*pi liefert nun 21.

Schreiben Sie das Beispiel ab und probieren Sie es aus!

- Ein Pointer kann nacheinander auf unterschiedliche Variablen des assoziierten Typs verweisen.
- Pointer kann man problemlos zuweisen.
- pi verweist per Initialisierung zunächst auf i, später wird die Adresse von j zugewiesen.

Pointer als Parameter

- Pointer können als Parameter an Funktionen übergeben werden.
- Wird ein Pointer an eine Funktion übergeben, wirkt sich eine Änderung des übegebenen Pointers nicht auf das Original aus. Eine Änderung an der durch den Pointer referenzierten Variable aber schon.
- Folgendes Beispiel demonstriert das:

```c
#include <stdio.h>

long fakult(int *i);

int main()
{
  int i;
  long f;

  for(i=0; i<10; i++)
  {
    f=fakult(&i);
    printf("Fakultaet von %d: %ld\n",i,f);
  }
  return 0;
}

long fakult(int* px)
{
  long f=1;
  int j;
  while (*px>1)f*=(*px)--;
  return f;
}
```

- Es handelt sich um das Beispiel Fakultät.
- An Stelle des Wertes von x wird die Adresse von x als Pointer übergeben.
- Entsprechend wird in der Funktion in Zeile 22 dereferenziert (*px).
- In main wird i aber dadurch 1, weil in Zeile 22 der dereferenzierte Wert dekrementiert wird, bis er 1 ist.

Pointer als Parameter

- Das Programm funktioniert nicht mehr, es läuft endlos (Abbrechen mit Strg-c).
- Beheben können wir dieses Verhalten, wenn wir eine neue Variable j einführen:

```
10    for(i=0; i<10; i++)
11    {
12      int j=i;
13      f=fakult(&j);
14      printf("i: %d, j:%d\n",i,j);
15      printf("Fakultaet von %d: %ld\n",i,f);
16    }
```

Pointer als Parameter

- Das Verhalten, das uns hier Probleme macht, kann auch gewollt sein.
- Über Pointer kann man auch einen Wert an den Aufrufer zurückgeben, unabhängig von return.
- Damit kann man mehr als einen Wert zurück-geben.

```c
1  #include <stdio.h>
2
3  void toUpper(char* pChar)
4  {
5    *pChar= *pChar & (~0x20);
6  }
7
8  int main()
9  {
10   char c='a';
11   toUpper(&c);
12   printf("%c\n", c);
13   return 0;
14 }
```

Groß und Kleinbuchstaben unterscheiden sich In einem bit. 0x1x xxxx (Kleinbuchstabe) und 0x0xxxxx (Großbuchstabe).
0X41: ‘A‘ und 0x61: ‘a‘
~0x20 (0010 0000) → 1101 1111 (bitweises not)
*pChar & (~0x20) verknüpft btweise den AsciiCode des Kleinbuchstaben mit der Maske und löscht somit das entscheidende Bit.

- Die Funktion toUpper erhält hier die Adresse eines Zeichens.
- In Zeile 5 wird mit *pChar=… ein Wert auf die Adresse, auf die der Pointer zeigt, zugewiesen. Er landet hier letztlich in der Variablen c in main.
- Damit ändert sich nun auch der Wert von c in main.

```c
#include <stdio.h>

void swap(int *pa, int *pb)
{
  *pa^=*pb; *pb^=*pa; *pa^=*pb;
}

int main()
{
  int i=12, j=99;
  printf("i:%d j:%d\n",i,j);
  swap(&i,&j);
  printf("i:%d j:%d\n",i,j);
  return 0;
}
```

```
$ ./a.out
i:12 j:99
i:99 j:12
```

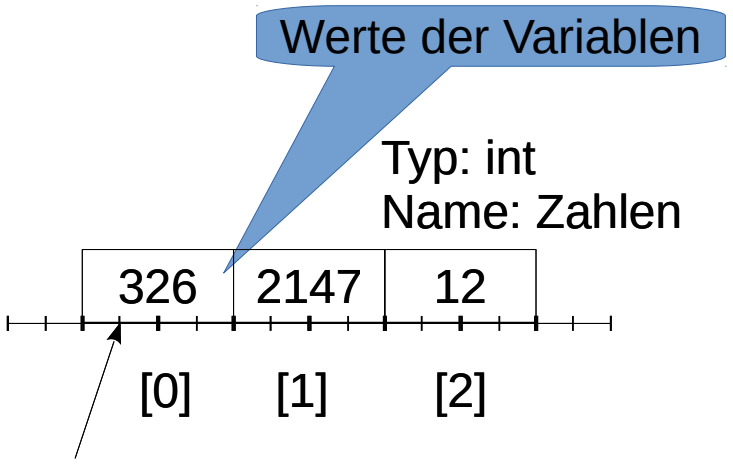
Schreiben Sie die Beispiele ab und probierens Sie sie aus!

- Die Funktion swap bekommt zwei Pointer auf int-Werte und vertauscht diese.
- Es werden auch die originalen Werte von i und j in main vertauscht. (siehe Ausgabe unten links)

Im Beispiel rechts funktioniert das nicht, weil nur die Kopien von i und j unter den Namen a und b vertauscht werden. i und j bleiben unberührt.

```c
#include <stdio.h>

void swap(int a, int b)
{
  a^=b; b^=a; a^=b;
}

int main()
{
  int i=12, j=99;
  printf("i:%d j:%d\n",i,j);
  swap(i,j);
  printf("i:%d j:%d\n",i,j);
  return 0;
}
```

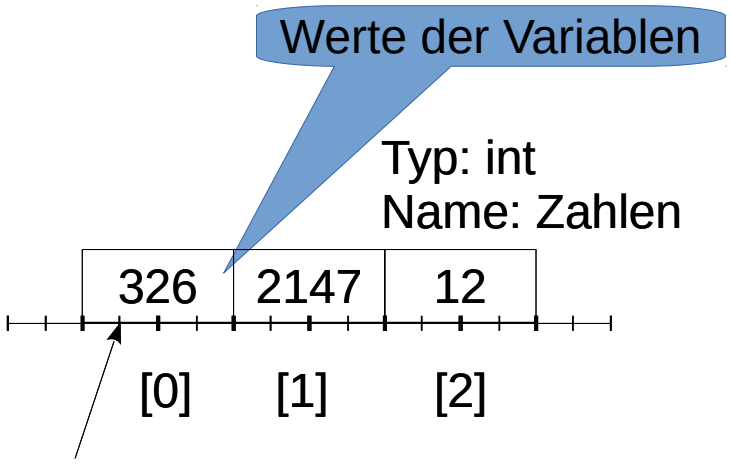
Pointer und Arrays




```
int array[3]={326,2147,12};
```

Wir haben bereits bei der Betrachtung primitiver Datentypen gelernt, dass der Name eines Arrays mit der Adresse des ersten Elements verknüpft ist. Der Name ist hier array und bezeichnet die niederwertigste Adresse des Wertes 326 innerhalb des Arrays.

Pointer und Arrays

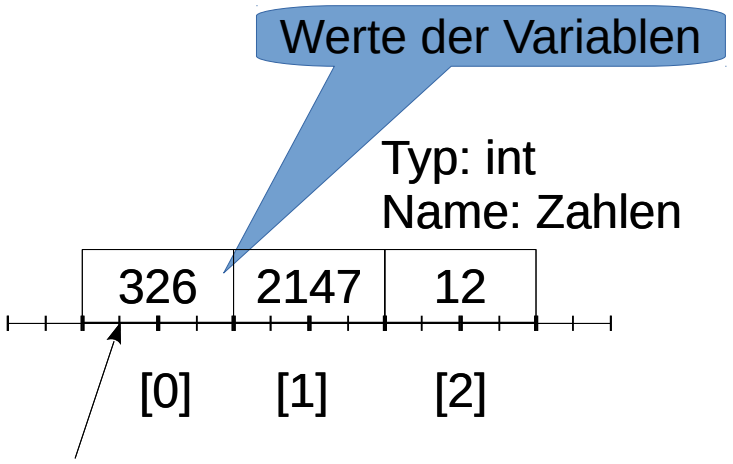


Adresse des Arrays

```
int array[3]={326,2147,12};
int * pi=array;
```

Diese Adresse des Arrays kann auch einer Pointervaria-blen zugewiesen werden. Da-bei wird kein Adressoperator verwendet, weil der Name ar-ray ja schon die Adresse eines int-Wertes verkörpert.

Pointer und Arrays



Adresse des Arrays

```
int array[3]={326,2147,12};
int * pi=array;
```

Da array lediglich ein konstanter Pointer auf einen int-Wert darstellt, können wir statt

array[0] auch

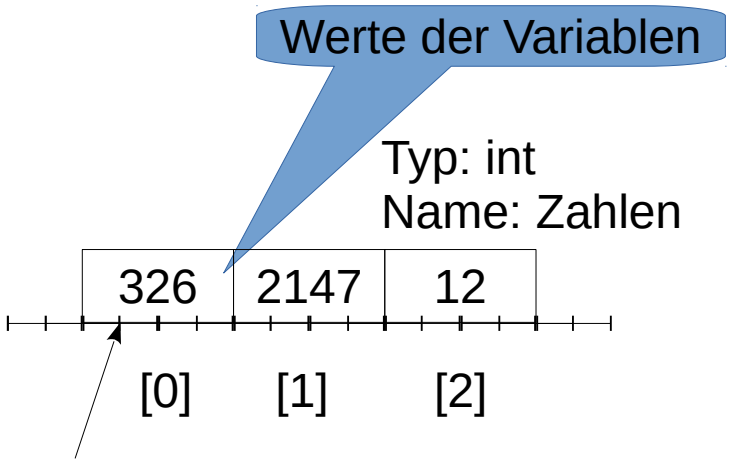
*array oder

*(array+0)

verwenden. Alle drei Notationen sind erlaubt und liefern 326.

array++ funktioniert aber nicht, da array ein konstanter Pointer ist.

Pointer und Arrays



Adresse des Arrays

```
int array[3]={326,2147,12};
int * pi=array;
```

Umgekehrt können wir auch statt array[0] verwenden:

*pi,

*(pi+0)

pi[0]

Alle drei Schreibweisen liefern auch den Wert 326.

```c
#include <stdio.h>



int main()
{
  int array[3]={326,2147,12};
  int * pi=array;

  printf("array[0]   : %d\n",array[0]);
  printf("*array     : %d\n",*array);
  printf("*(array+0): %d\n",*(array+0));
  printf("*pi        : %d\n",*pi);
  printf("*(pi+0)    : %d\n",*(pi+0));
  printf("pi[0]      : %d\n",pi[0]);

  return 0;
}
```

```
$ ./a.out
array[0]   : 326
*array     : 326
*(array+0): 326
*pi        : 326
*(pi+0)    : 326
pi[0]      : 326
```

Pointerarithmetik

- Mit Pointern kann man auch rechnen, aber sie folgen einer eigenen Arithemtik.
- Erlaubt sind die Addition und die Subtraktion ganzzahliger Werte zu Pointern.
- Dabei werden die zu addierenden oder zu subtrahierenden Werte zunächst mit der Größe in Bytes des assoziierten Datentyps multipliziert, erst dann wird die Addition durchgeführt.

Pointerarithmetik

- Bezogen auf unser letztes Beispiel, in dem es das Array array von int-Werten gab, bedeutet das:
  - pi+1 liefert pi+1*sizeof(int)
  - Mit sizeof(int)=4, ein int-Wert belegt 4 Byte im Speicher, wird die Adresse bei einer Addition von 1 um den Wert 4 erhöht, bei double mit einer Länge von 8 Byte, entsprechend 8.
- Pointeraritmetik rechnet immer mit Items des assoziiertern Datentyps.

```c
#include <stdio.h>



int main()
{
  int array[3]={326,2147,12};
  int * pi=array;

  printf("array[1]   : %d\n",array[1]);
  printf("*(array+1): %d\n",*(array+1));
  printf("*(pi+1)    : %d\n",*(pi+1));
  printf("pi[1]      : %d\n",pi[1]);

  return 0;
}
```

```
$ ./a.out
array[1]   : 2147
*(array+1): 2147
*(pi+1)    : 2147
pi[1]      : 2147
```

Im nebenstehenden Beispiel wird der Zugriff auf das Element mit dem Index 1 auf verschiedene Weise demonstriert.

```c
#include <stdio.h>

int main()
{
  int array[3]={326,2147,12};
  int * pi=array;
  int i;
  printf("array[i]    :");
  for (i=0; i<sizeof(array)/sizeof(int); i++)
    printf("%d, ",array[i]);
  printf("\n");
  printf("pi[i]       :");
  for (i=0; i<sizeof(array)/sizeof(int); i++)
    printf("%d, ",pi[i]);
  printf("\n");
  printf("*(array+i)  :");
  for (i=0; i<sizeof(array)/sizeof(int); i++)
    printf("%d, ",*(array+i));
  printf("\n");
  printf("*(pi+i)     :");
  for (i=0; i<sizeof(array)/sizeof(int); i++)
    printf("%d, ",*(pi+i));
  printf("\n");
  printf("*pi++       :");
  for (i=0; i<sizeof(array)/sizeof(int); i++)
    printf("%d, ",*pi++);
  printf("\n");

  return 0;
}
```

Das nebenstehende Beispiel demonstriert abschließend die verschiedenen Schreibweisen.

```
$ ./a.out
array[i]    :326, 2147, 12,
pi[i]       :326, 2147, 12,
*(array+i)  :326, 2147, 12,
*(pi+i)     :326, 2147, 12,
*pi++       :326, 2147, 12,
```

Operationen mit Pointern

- Zuweisung
- Vergleich auf == und !=
- Addition von int-Werten
- Subtraktion von int-Werten
- Pointer lassen sich nach long konvertieren
- Dereferenzierung *
- Asdressbildung &

Generischer Pointern

- Ein Pointer vom Typ void* wird auch „Generischer Pointer“ bezeichnet.
- Er ist zuweisungskompatibel zu jedem Pointer.
- Er kann nicht dereferenziert werden.
- Mit void-Pointern kann man nicht rechnen.

NULL

- NULL ist ein spezieller Pointer, der Nullpointer.
- Dahinter verbirgt sich
- (void*)0L
- Mittels Typecast wird der Longwert 0 in einen ungetypten Pointer (generischer Pointer) umgewandelt.

Pointer/Arrays/Funktionen

- Da Arrays durch die Adresse des ersten Elementes repräsentiert werden, wird bei Parameterübergaben von Arrays an Funktionen nur die Adresse des ersten Elementes übergeben.
- Eine Information über die Länge des Arrays gibt es dabei nicht.
- Die Anzahl der Arrayelemente sollte deshalb immer als zusätzliches Argument (zusätzlicher Parameter) übergeben werden.
- Ausnahme bilden Zeichenketten, deren Länge durch die terminierende 0 erkannt werden kann.

```c
#include <stdio.h>

void dispArray(int * pdata, int n)
{
    int i;
    for (i=0; i<n; i++)
        printf("%d, ",*(pdata+i));
    printf("\n");
}

int main()
{
  int array[3]={326,2147,12};
  int * pi=array;

  dispArray(array,sizeof array / sizeof(int));
  return 0;
}
```

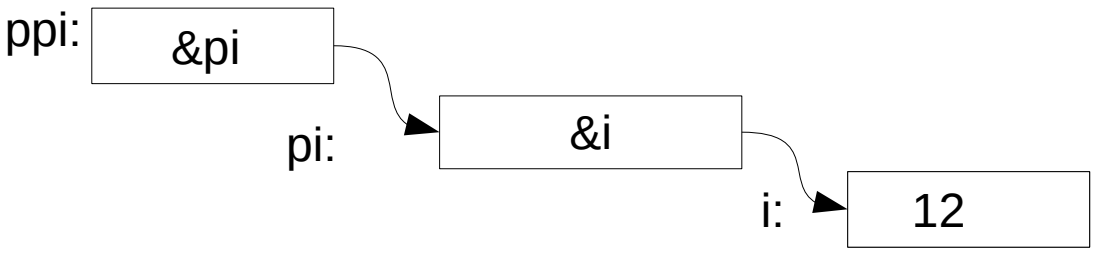
```
./a.out
326, 2147, 12,
```

Größe des Arrays geteilt
Durch die Größe eines
Arrayelementes ergibt
die Anzahl der Arrayelemente

- An die Funktioon dispArray wird das Array array übergeben.
- Es wird aber nur die Adresse des ersten Elementes übergeben (pdata).
- Über den zweiten Parameter int n wird die Größe des Arrays übergeben.

Doppelpointer

- Ein Pointer kann auch wiederum auf einen Pointer zeigen.




```
int  i   =12;
int *pi  =&i;
int**ppi =&pi;
```

CommandLineArguments

- Eine Konstruktion dieser Form gibt es als Parameter der main-Funktion als sog. Commandlinearguments oder dt. Kommandozeilenargumente.
- Sie ermöglichen, Werte vom Programmaufruf an ein c-Programm zu übergeben.

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

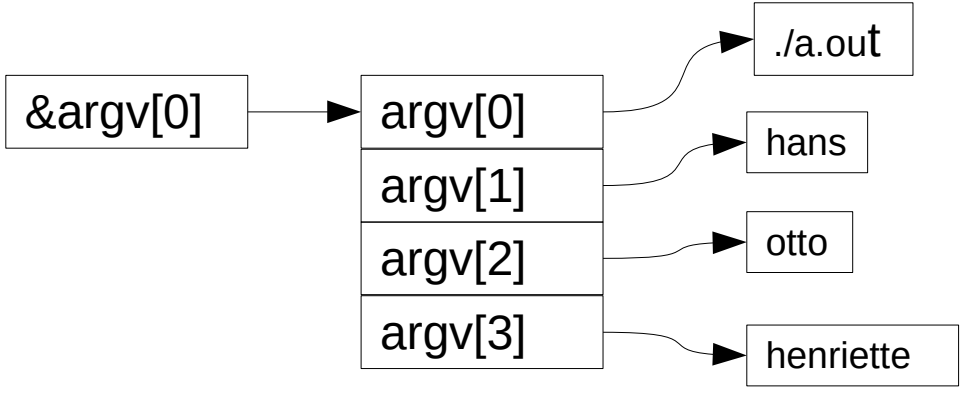
- Dabei enthält argc die Anzahl der Parameter und argv die Argumente als Strings oder nullterminierte Zeichenketten.

```c
#include <stdio.h>

int main(int argc, char**argv)
{
  int i;
  for (i=0; i<argc; i++)
      printf("argv[%d]: %s\n",i,argv[i]);

  return 0;
}
```

```
$ ./a.out
argv[0]: ./a.out
```

```
$ ./a.out hans otto anna henriette
argv[0]: ./a.out
argv[1]: hans
argv[2]: otto
argv[3]: anna
argv[4]: henriette
```

- Das erste Commandlienarg. ist in c immer der Programmaufruf selbst.
- Die weiteren Argumente folgen.
- argv ist ein Pointer auf ein Array von char-Pointern, die auf Strings verweisen.
- Die Schreibweise char* argv[] ist eben-falls möglich und auch üblich.



CommandLineArguments

- Bei der Verwendung von Kommandozeilenargumenten sollte als erstes in main getestet werden, ob die nötigen Argumente beim Programmaufruf angegeben worden sind.

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

int main(int argc, char**argv)
{
  if (argc !=3){printf("usage: %s <name1> <name2>\n",argv[0]); exit (-1);}

  // alles ok, zwei Namen wurden angegeben
  printf("%s liebt %s\n",argv[1],argv[2]);

  return 0;
}
```

Hier werden die Kommandozeilenargumente getestet, bei Bedarf wird eine usage-Meldung ausgegeben.

Programm nicht Erfolgreich benden.

```
$ ./a.out
usage: ./a.out <name1> <name2>
$ ./a.out hans anna
hans liebt anna
$
```

Pointer als Returnwert

- Eine Funktion kann einen Pointer zurückgeben.
- Zu beachten ist dabei, dass die Variable, auf die der Pointer verweist, noch existiert, wenn die Funktion verlassen wurde.
- Eine Funktion darf keinen Pointer auf automatische, lokale Variable zurückgeben.
- Pointer auf lokale statische Variablen dürfen zurückgegeben werden, da diese bis zum Ende der Programmausführung existieren.