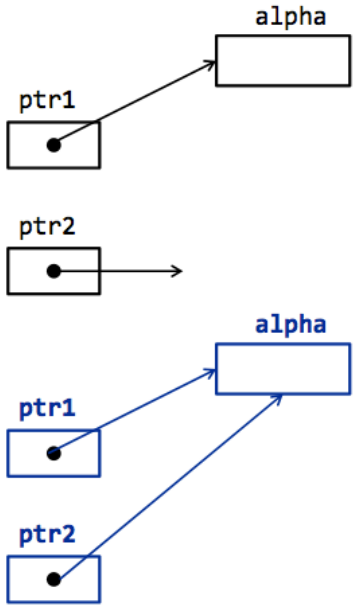
Darstellung in graphischer Pointernotation

```
float alpha;

float* ptr1 = &alpha;

float* ptr2;

ptr2 = ptr1;
```



© HSR Prof. R. Bonderer

Beispiel: Character-Array

```
/* Datei: char_array.c */
/* Entscheiden, ob eine Zeichenkette ein 'a' enthält              */
#include <stdio.h>
#include <string.h>
#define MAX 40

int main(void)
{
  int index;
  char eingabe[MAX+1];
  printf("Bitte String eingeben (max. %d Zeichen): ", MAX);
  gets(eingabe);                                                  /* 1 */
  for (index = 0; eingabe[index] != '\0'; index = index + 1)
  {
    if (eingabe[index] == 'a')                                    /* 2 */
      break;                                                      /* 3 */
  }
  if (eingabe[index] == '\0')                                     /* 4 */
  {
    printf("\nIhr String enthaelt kein 'a'\n");
  }
  else                                                            /* 5 */
  {
    printf("\nDas 'a' befand sich an der %d. Stelle\n", index + 1);
  }
  return 0;
}
```

© HSR Prof. R. Bonderer

Beispiel: arraypar.c

Parameter ist Pointer auf Arrayelement

```
enum {groesse = 3};

void init(int* alpha, int dim);
void ausgabe(int alpha[], int dim);

int main(void)
{
  int arr[groesse];
  init(arr, groesse);
  ausgabe(arr, groesse);
  return 0;
}
```

Parameter ist offenes Array

Argument ist Name des Arrays

© HSR Prof. R. Bonderer


```
/* Datei: arraypar.c */
#include <stdio.h>
enum {groesse = 3};

void init(int* alpha, int dim);
void ausgabe(int alpha[], int dim);

int main(void)
{
  int arr[groesse];
  init(arr, groesse);
  ausgabe(arr, groesse);
  return 0;
}

void init(int* alpha, int dim)   /* hier ist alpha ein Pointer   */
{                                /* auf den Elementtyp eines Arrays */
  int i;
  for (i = 0; i < dim; i++)
  {
    printf("Eingabe Komponente mit Index %d von arr: ", i);
    scanf("%d", alpha++);
  }
}

void ausgabe(int alpha[], int dim)  /* hier ist alpha vom Typ     */
{                                   /* eines offenen Arrays       */
  int i;
  for (i = 0; i < dim; i++)
  {
    printf("\narr[%d] hat den Wert: %d", i, alpha[i]);
  }
}
```

© HSR Prof. R. Bonderer

const bei Pointern: konstanter String

1. Variante

```
char str[] = "Ein String";
const char* text = str;
```

Dies bedeutet nicht, dass der Pointer `text` konstant ist, sondern dass `text` auf einen konstanten String zeigt.

Von rechts nach links lesen:
"text ist ein Pointer auf eine char-Konstante"

```
char ch = text[1];              // erlaubt (== 'i')
text[1] = 's';                  // nicht erlaubt
text = "Ein anderer String";    // erlaubt
str[4] = 'A';                   // erlaubt
```

© HSR Prof. R. Bonderer

const bei Pointern: konstanter Pointer

2. Variante

```
char str[] = "Ein String";
char* const text = str;
```

Hier ist nun der Pointer `text` konstant. Die Position von `const` ist sehr relevant!

Von rechts nach links lesen:
"text ist ein konstanter Pointer auf ein char"

```
char ch = text[1];              // erlaubt (== 'i')
text[1] = 's';                  // erlaubt
text = "Ein anderer String";    // nicht erlaubt
str[4] = 'A';                   // erlaubt
```

© HSR Prof. R. Bonderer

const bei Pointern: konstanter Pointer auf konstanten String

3. Variante

```
char str[] = "Ein String";
const char* const text = str;
```

Bei dieser Variante ist sowohl der Pointer `text` als auch der String, auf welchen `text` zeigt, konstant.

Von rechts nach links lesen:
"text ist ein konstanter Pointer auf eine char-Konstante"

```
char ch = text[1];              // erlaubt (== 'i')
text[1] = 's';                  // nicht erlaubt
text = "Ein anderer String";    // nicht erlaubt
str[4] = 'A';                   // erlaubt
```

© HSR Prof. R. Bonderer

Mit Array: strcopy1.c

```
#include <stdio.h>

int main(void)
{
  char alpha[30] = "zu kopierender String";
  char beta[30];

  int i = 0;

  while (alpha[i] != '\0')
  {
    beta[i] = alpha[i];
    i++;
  }
  beta[i] = '\0';

  printf("alpha = %s\n", alpha);
  printf("beta  = %s\n", beta);
}
```

© HSR Prof. R. Bonderer

Mit Pointer: strcopy6.c

```
#include <stdio.h>

int main(void)
{
  char alpha[30] = "zu kopierender String";
  char beta[30];
  char* palpha = alpha;  // brauchts, da alpha konstanter Pointer
  char* pbeta = beta;    // dito beta

  while (*pbeta++ = *palpha++)
  {
    // nichts zu tun
  }

  printf("alpha = %s\n", alpha);
  printf("beta  = %s\n", beta);
}
```

© HSR Prof. R. Bonderer

Beispiel: Stringtabelle mit offenem Array (strTable1.c)

```
/* Datei: strTable1.c */
#include <stdio.h>

void textAusgabe(char* textPointer[], int zeilen)
{
  int i;
  for (i = 0; i < zeilen; i++)
  {
    printf("%s\n", textPointer[i]);
  }
}

int main(void)
{
  char* strTable[] = {"Pflaume",
                      "Apfel",
                      "Johannisbeere"};

  textAusgabe(strTable, 3);
  return 0;
}
```

© HSR Prof. R. Bonderer

Beispiel für Funktionspointer (ftptr.c)

```c
#include <stdio.h>

int foo(char ch)
{
  int i;
  for (i = 1; i <= 10; i++)
    printf("%c ", ch);
  printf("\n");
  return i;
}

int main(void)
{
  int (*p)(char);   // Deklaration des Funktionspointers
  char c;
  int ret;
  printf("Buchstabe eingeben: ");
  scanf("%c", &c);
  p = foo;          // ermittle Adresse der Funktion foo()
  ret = p(c);       // Aufruf von foo() über Funktionspointer
  return 0;
}
```

© HSR Prof. R. Bonderer

Beispiel: Stringtabelle mit Pointer auf Pointer (strTable2.c)

```c
/* Datei: strTable2.c */
#include <stdio.h>

void textAusgabe(char** textPointer, int zeilen)
{
  int i;
  for (i = 0; i < zeilen; i++)
    printf("%s\n", *textPointer++);
}

int main(void)
{
  char* strTable[] = {"Pflaume",
                      "Apfel",
                      "Johannisbeere"};

  textAusgabe(strTable, 3);
  return 0;
}
```

© HSR Prof. R. Bonderer

Fakultät: iterative vs. rekursive Implementation

Fakultät iterativ

```
unsigned long
faku(unsigned int n)
{
  unsigned long fak = 1UL;
  unsigned int i;
  for (i = 2; i <= n; ++i)
  {
    fak = fak * i;
  }
  return fak;
}
```

Fakultät rekursiv

```
unsigned long
faku(unsigned int n)
{
  if (n > 1)
  {
    return n * faku(n-1);
  }
  else
  {
    return 1UL;
  }
}
```

© HSR Prof. R. Bonderer

Beispiel: typdef.c

```
typedef enum {black, red, green, yellow, blue} Color;

typedef struct
{
  int x;
  int y;
} Point;

typedef struct
{
  Point p1;
  Point p2;
  Color col;
} Line;

int main (void)
{
  Line myLine = {{12, -34},    // p1
                 {783, 12},    // p2
                 yellow};      // col
  printf ("Anfangspunkt: (%d, %d)\n", myLine.p1.x, myLine.p1.y);
  printf ("Endpunkt:     (%d, %d)\n", myLine.p2.x, myLine.p2.y);
  printf ("Farbe:        %d\n", myLine.col);
  return 0;
}
```

keine Klammern notwendig

© HSR Prof. R. Bonderer

makefile

```
# makefile
cc=gcc
CFLAGS= -c -Wall
LFLAGS= -Wall
OBJ=foo.o goo.o main.o
EXE=prog.exe

${EXE}: ${OBJ}
        ${cc} ${LFLAGS} -o ${@} ${OBJ}

foo.o:  foo.h goo.h foo.c
        ${cc} ${CFLAGS} -o ${@} foo.c

goo.o:  goo.h goo.c
        ${cc} ${CFLAGS} -o ${@} goo.c

main.o: foo.h main.c
        ${cc} ${CFLAGS} -o ${@} main.c

clean:
        rm -f ${OBJ} ${EXE}
```

Targets: Diese können mit make angesprungen werden, z.B. make goo.o

Abhängigkeitsliste, d.h. von diesen Dateien ist das Target abhängig

Befehl, der ausgeführt werden muss, falls etwas geändert hat. Wichtig: erstes Zeichen muss ein TAB sein.

Erstellt foo

remove

Löscht alle alten files

© HSR Prof. R. Bonderer

Binäre Ein- / Ausgabe

- Die binäre Ein- und Ausgabe wird dann angewendet, wenn Daten gespeichert werden sollen, die nicht in lesbarer Form vorliegen. Die binäre Datei muss zwingend im Binärmodus geöffnet werden.

```
#include <stdio.h>
size_t fwrite(const void* ptr, size_t size,
              size_t nmemb, FILE* stream);
// schreibt nmemb Objekte der Grösse size aus dem Array,
// auf das ptr zeigt in stream


size_t fread(void* ptr, size_t size,
             size_t nmemb, FILE* stream);
// analog für Schreiben
```

Demo: `fwrite2.c`

© HSR Prof. R. Bonderer