C-Kurs
Pointers

Sebastian@Pipping.org

22. September 2009

C-Kurs

	Mo	Konzepte, Syntax, ... printf, scanf
Next →	Di	Pointers, Arrays, ... Structs, malloc, ...
	Mi	Compiler, Headers, ... Debugging I
	Do	Guter Code Debugging II
	Fr	stdlib, Bücher, ...

Übersicht

- Pointer
- Arrays
- Strings

Übersicht

Grundlagen für ..

- Multiplikation von Matrizen
- Speicherverwaltung
  (nächste Vorlesung)

Themen

- Hello pointer
- Pointer auf Pointer
- Arrays und Pointer
- Pointer-Arithmetik
- Strings und Längenberechnung
- const correctness
- argc und argv
- Mehrdimensionale Arrays
- Deklaration von Strings
- NULL-Pointer

- Weiterführende Themen
- Zusammenfassung

Pointer

Pointer

- zeigen auf eine Stelle im Speicher

Pointer

- zeigen auf eine Stelle im Speicher
- sind Adressen

Pointer

- zeigen auf eine Stelle im Speicher
- sind Adressen
- sind Zahlen

Pointer

- zeigen auf eine Stelle im Speicher
- sind Adressen
- sind Zahlen
- sind quasi auch Referenzen

Hello pointer

Hello pointer

```
#include <stdio.h>

int
main() {









  return 0;
}
```

Hello pointer

```
#include <stdio.h>

int
main() {
  int i = 3;









  return 0;
}
```

Hello pointer

```
#include <stdio.h>

int
main() {
  int i = 3;
  int *p = &i;









  return 0;
}
```

Hello pointer

```
#include <stdio.h>

int
main() {
  int i = 3;
  int *p = &i;

  printf(" p == %#p\n"
         "*p == %d\n"
         " i == %d\n", p, *p, i);



  return 0;
}
```

Hello pointer

```
#include <stdio.h>

int
main() {
  int i = 3;
  int *p = &i;

  printf(" p == %#p\n"
         "*p == %d\n"
         " i == %d\n", p, *p, i);

  *p = 14;
  printf(" i == %d\n\n", i);
  return 0;
}
```

Ausgabe

```
# gcc hello_pointer.c -o hello_pointer
```

Ausgabe

```
# gcc hello_pointer.c -o hello_pointer
# ./hello_pointer
```

Ausgabe

```
# gcc hello_pointer.c -o hello_pointer
# ./hello_pointer
 p == 0xbfa3ad8c
*p == 3
 i == 3
 i == 14
```

Ausgabe

```
# gcc hello_pointer.c -o hello_pointer
# ./hello_pointer
 p == 0xbfa3ad8c
*p == 3
 i == 3
 i == 14

# ./hello_pointer
 p == 0xbfba66fc
*p == 3
 i == 3
 i == 14
```

Ausgabe

```
# gcc hello_pointer.c -o hello_pointer
# ./hello_pointer
 p == 0xbfa3ad8c
*p == 3
 i == 3
 i == 14

# ./hello_pointer
 p == 0xbfba66fc  ← Andere Adresse!
*p == 3
 i == 3
 i == 14
```

Zusammenfassung 1/2

Pointer zeigen auf typisierte Daten:

`int *` == Pointer auf int

`char *` == Pointer auf char

Zusammenfassung 2/2

- Prefix-Operator & liefert Adresse

Zusammenfassung 2/2

- Prefix-Operator & liefert Adresse
- Prefix-Operator * dereferenziert

Zusammenfassung 2/2

- Prefix-Operator & liefert Adresse
- Prefix-Operator * dereferenziert
- & und * sind komplementär:

  *(&x) == x

Pointer auf Pointer

Pointer auf Pointer

```
int i = 3;
int *p = &i;
```

Pointer auf Pointer

```
int i = 3;
int *p = &i;
int **pp = &p;
```

Wohin mit den Sternen? (1/2)

```
int**  foo;
int ** foo;
int  **foo;
```

Wohin mit den Sternen? (1/2)

```
int**  foo;
int ** foo;
int  **foo;
```

← Bitte nicht!

Wohin mit den Sternen? (1/2)

```
int * foo;
int ** foo;
int   **foo;
```

Wohin mit den Sternen? (2/2)

```
int x, y, z;
```

Wohin mit den Sternen? (2/2)

```
int x, y, z;

int** a, b;
```

Wohin mit den Sternen? (2/2)

```
int x, y, z;

int** a, b;  /* Misleading */
```

Wohin mit den Sternen? (2/2)

```
int x, y, z;

int** a, b;  /* Misleading */
int **a, b;
```

Wohin mit den Sternen? (2/2)

```
int x, y, z;

int** a, b;  /* Misleading */
int **a, b;  /* Better */
```

Wohin mit den Sternen? (2/2)

```
int x, y, z;

int** a, b;  /* Misleading */
int **a, b;  /* Better */

/* Best: */
int **a;
int b;
```

Arrays und Pointer

Arrays und Pointer (1/3)

```
#include <stdio.h>

int
main() {









  return 0;
}
```

Arrays und Pointer (1/3)

```
#include <stdio.h>

int
main() {
  int primes[] = {2, 3, 5, 7, 11};




  return 0;
}
```

Arrays und Pointer (1/3)

```
#include <stdio.h>

int
main() {
  int primes[] = {2, 3, 5, 7, 11};
  int i;




  return 0;
}
```

Arrays und Pointer (1/3)

```
#include <stdio.h>

int
main() {
  int primes[] = {2, 3, 5, 7, 11};
  int i;

  for (i = 0; i < 5; ++i) {


  }
  return 0;
}
```

Arrays und Pointer (1/3)

```
#include <stdio.h>

int
main() {
  int primes[] = {2, 3, 5, 7, 11};
  int i;

  for (i = 0; i < 5; ++i) {
    printf("primes[%d] == %d\n", i,
        primes[i]);
  }
  return 0;
}
```

Zusammenfassung

Zugriff via [*n*] ähnlich Java

Arrays und Pointer (2/3)

```
#include <stdio.h>

int
main() {
  int primes[] = {2, 3, 5, 7, 11};
  int i;

  for (i = 0; i < 5; ++i) {
    printf("primes[%d] == %d\n", i,
        primes[i]);
  }
  return 0;
}
```

Arrays und Pointer (2/3)

```
#include <stdio.h>

int
main() {
  int primes[] = {2, 3, 5, 7, 11};
  int i;

  for (i = 0; i < sizeof(primes) / sizeof(int); ++i) {
    printf("primes[%d] == %d\n", i,
        primes[i]);
  }
  return 0;
}
```

Zusammenfassung (1/2)

`sizeof(`*`variable`*`)`

==

Von *`variable`* belegter Speicher in Byte

Zusammenfassung (2/2)

`sizeof(`*`type`*`)`

==

Von *type*-Instanzen belegter Speicher in Byte

Arrays und Pointer (3/3)

```
#include <stdio.h>

int
main() {
  int primes[] = {2, 3, 5, 7, 11};
  int i;

  for (i = 0; i < sizeof(primes) / sizeof(int); ++i) {
    printf("primes[%d] == %d\n", i,
        primes[i]);
  }
  return 0;
}
```

Arrays und Pointer (3/3)

```
#include <stdio.h>

int
main() {
  int primes[] = {2, 3, 5, 7, 11};
  int i;

  for (i = 0; i < sizeof(primes) / sizeof(int); ++i) {
    printf("primes[%d] == %d\n", i,
        primes[i]);


  }
  return 0;
}
```

Arrays und Pointer (3/3)

```
#include <stdio.h>

int
main() {
  int primes[] = {2, 3, 5, 7, 11};
  int i;

  for (i = 0; i < sizeof(primes) / sizeof(int); ++i) {
    printf("primes[%d] via array   == %d\n", i,
        primes[i]);


  }
  return 0;
}
```

Arrays und Pointer (3/3)

```
#include <stdio.h>

int
main() {
  int primes[] = {2, 3, 5, 7, 11};
  int i;

  for (i = 0; i < sizeof(primes) / sizeof(int); ++i) {
    printf("primes[%d] via array   == %d\n", i,
        primes[i]);
    printf("primes[%d] via pointer == %d\n", i,
        *(primes + i));
  }
  return 0;
}
```

Arrays und Pointer (3/3)

```
#include <stdio.h>

int
main() {
  int primes[] = {2, 3, 5, 7, 11};
  int i;

  for (i = 0; i < sizeof(primes) / sizeof(int); ++i) {
    printf("primes[%d] via array   == %d\n", i,
        primes[i]);
    printf("primes[%d] via pointer == %d\n", i,
        *(primes + i));
  }
  return 0;
}
```

Zusammenfassung (1/2)

Array == Pointer auf den Anfang des Arrays

Zusammenfassung (2/2)

$$a[n] == *(a + n)$$

Pointer-Arithmetik

Pointer-Arithmetik

```
int numbers[3];
char text[] = "software libre";
```

Pointer-Arithmetik

```
int numbers[3];
char text[] = "software libre";
```

Es gilt:

```
numbers[i] == *(numbers + i)
```

Pointer-Arithmetik

```
int numbers[3];
char text[] = "software libre";
```

Es gilt:

```
numbers[i] == *(numbers + i)
text[j] == *(text + j)
```

Pointer-Arithmetik

```
int numbers[3];
char text[] = "software libre";
```

Es gilt:

```
numbers[i] == *(numbers + i)
text[j] == *(text + j)
sizeof(char) != sizeof(int)
```

Pointer-Arithmetik

```
int numbers[3];
char text[] = "software libre";
```

Es gilt:

```
numbers[i] == *(numbers + i)
text[j] == *(text + j)
sizeof(char) != sizeof(int)
```

Für Operatoren +/- auf Pointern folgt:

Pointer-Arithmetik

```
int numbers[3];
char text[] = "software libre";
```

Es gilt:

```
numbers[i] == *(numbers + i)
text[j] == *(text + j)
sizeof(char) != sizeof(int)
```

Für Operatoren +/- auf Pointern folgt:

Sprungweite variiert mit dem Typen!

Strings

Strings

"ABC"

Strings

"ABC" $\rightarrow$ {65, 66, 67, <u>00</u>}

Strings

"ABC" $\rightarrow \{65, 66, 67, \underline{00}\}$

```
#include <string.h>
...
strlen("ABC");
```

Strings

"ABC" $\rightarrow \{65, 66, 67, \underline{00}\}$

```
#include <string.h>
...
strlen("ABC");
```

$\rightarrow 3$

my_strlen

```
unsigned int
my_strlen(const char *str) {




  return                ;
}
```

my_strlen

```
unsigned int
my_strlen(const char *str) {
  const char *walker = str;



  return                ;
}
```

my_strlen

```
unsigned int
my_strlen(const char *str) {
  const char *walker = str;
  while ('\0' != *walker) {
    walker++;
  }
  return                ;
}
```

my_strlen

```
unsigned int
my_strlen(const char *str) {
  const char *walker = str;
  while (*walker) {
    walker++;
  }
  return                ;
}
```

my_strlen

```
unsigned int
my_strlen(const char *str) {
  const char *walker = str;
  while ('\0' != *walker) {
    walker++;
  }
  return                ;
}
```

my_strlen

```
unsigned int
my_strlen(const char *str) {
  const char *walker = str;
  while ('\0' != *walker) {
    walker++;
  }
  return walker - str;
}
```

Strings

```
strlen("ABC");
```

Strings

```
strlen("ABC");
```

$\longrightarrow$ 3

Strings

```
strlen("ABC");
→ 3

strlen("A\0B\0C");
```

Strings

```
strlen("ABC");
→ 3

strlen("A\0B\0C");
→ 1
```

Strings

```
strlen("ABC");
→ 3

strlen("A\0B\0C");
→ 1

strlen("ABC\0\0");
```

Strings

```
strlen("ABC");
→ 3

strlen("A\0B\0C");
→ 1

strlen("ABC\0\0");
→ 3
```

const correctness

const correctness (1/7)

```
char const * const foo = "foo";
```

const correctness (1/7)

```
char const * const foo = "foo";
```

const correctness (2/7)

```
_____ int _____ foo;
```

const correctness (3/7)

```
int const  x = 3;

const int  y = 4;
```

const correctness (4/7)

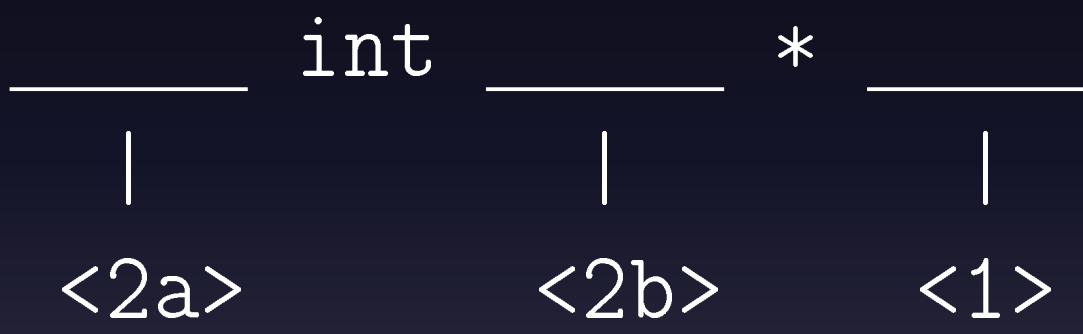
```
_____ int _____ foo;
```

const correctness (4/7)

```
_____ int _____ * _____ foo;
```

const correctness (4/7)

const correctness (4/7)



const correctness (5/7)

```
<1>  int * const foo;
```

const correctness (5/7)

```
<1>  int * const foo;
```

Verbietet:

```
foo = ...;
```

const correctness (5/7)

```
<1>  int * const foo;
```

Verbietet:

```
foo = ...;
```

```
<2a> const int *foo;
<2b> int const *foo;
```

const correctness (5/7)

<1> `int * const foo;`

Verbietet:

`foo = ...;`

<2a> `const int *foo;`

<2b> `int const *foo;`

Verbietet:

`foo[3] = ...;`

const correctness (6/7)

```
<2a,1> int const * const foo;
```

const correctness (6/7)

```
<2a,1> int const * const foo;
```

Verbietet:

```
foo = ...;
foo[5] = ...;
```

const correctness (6/7)

```
<2a,1> int const * const foo;
```

Verbietet:

```
foo = ...;
foo[5] = ...;
```

String-Konstanten bitte immer so!

const correctness (6/7)

```
<2a,1> int const * const foo;
```

Verbietet:

```
foo = ...;
foo[5] = ...;
```

String-Konstanten bitte immer so!

```
char const * const foo = "foo";
```

const correctness (7/7)

```
int const * const * const foo;
```

const correctness (7/7)

```
int const * const * const foo;
```

Verbietet:

```
foo = ...;
foo[7] = ...;
foo[2][4] = ...;
```

const correctness in APIs

```
int my_strlen(char *str);
```

```
int my_strlen(const char *str);
```

const correctness in APIs

```
int my_strlen(char *str);
```

Funktion darf *Inhalt* von `str` verändern

```
int my_strlen(const char *str);
```

const correctness in APIs

```
int my_strlen(char *str);
```

Funktion darf *Inhalt* von `str` verändern

my_strlen("ABC") gibt Compile-Fehler

```
int my_strlen(const char *str);
```

const correctness in APIs

```
int my_strlen(char *str);
```

Funktion darf *Inhalt* von `str` verändern

my_strlen("ABC") gibt Compile-Fehler

```
int my_strlen(const char *str);
```

Funktion darf Inhalt von `str` *nicht* verändern

const correctness in APIs

```
int my_strlen(char *str);
```

Funktion darf *Inhalt* von `str` verändern

my_strlen("ABC") gibt Compile-Fehler

```
int my_strlen(const char *str);
```

Funktion darf Inhalt von `str` *nicht* verändern

my_strlen("ABC") erlaubt und sicher

argc und argv

argc und argv

```
#include <stdio.h>

int
main() {




  return 0;
}
```

argc und argv

```
#include <stdio.h>

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




  return 0;
}
```

argc und argv

```
#include <stdio.h>

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

  printf("%d parameters\n", argc - 1);



  return 0;
}
```

argc und argv

```
#include <stdio.h>

int
main(int argc, char **argv) {
  int i = 0;
  printf("%d parameters\n", argc - 1);
  for (; i < argc; ++i) {

  }
  return 0;
}
```

argc und argv

```
#include <stdio.h>

int
main(int argc, char **argv) {
  int i = 0;
  printf("%d parameters\n", argc - 1);
  for (; i < argc; ++i) {
    printf("[%d] %s\n", i, argv[i]);
  }
  return 0;
}
```

Ausgabe

```
# gcc print_args.c -o print_args
```

Ausgabe

```
# gcc print_args.c -o print_args
# ./print_args free 'open source' software
```

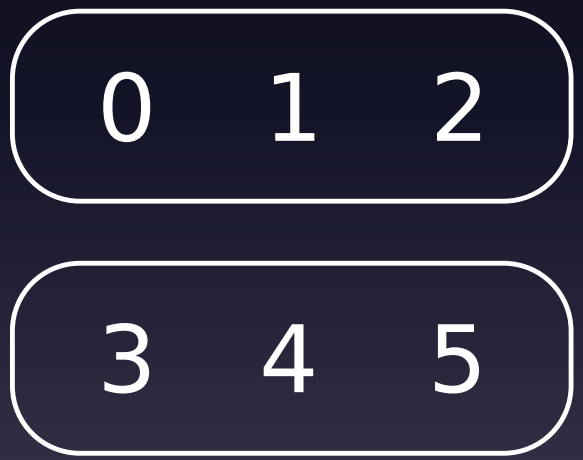
Ausgabe

```
# gcc print_args.c -o print_args
# ./print_args free 'open source' software
3 parameters
[0] ./print_args
[1] free
[2] open source
[3] software
```

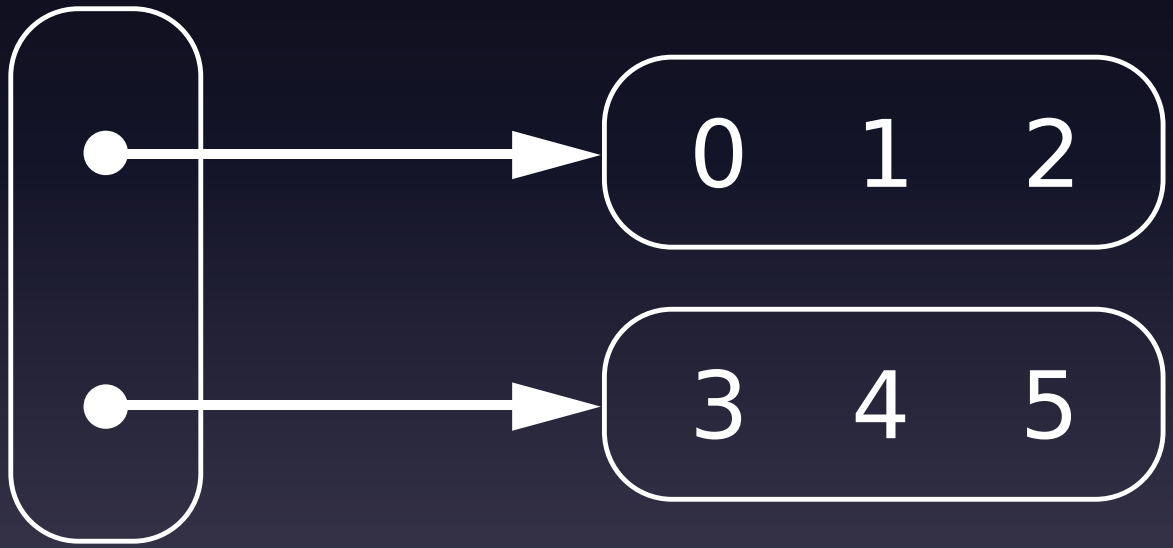
Mehrdimensionale Arrays

Variante
"Array von Pointern"

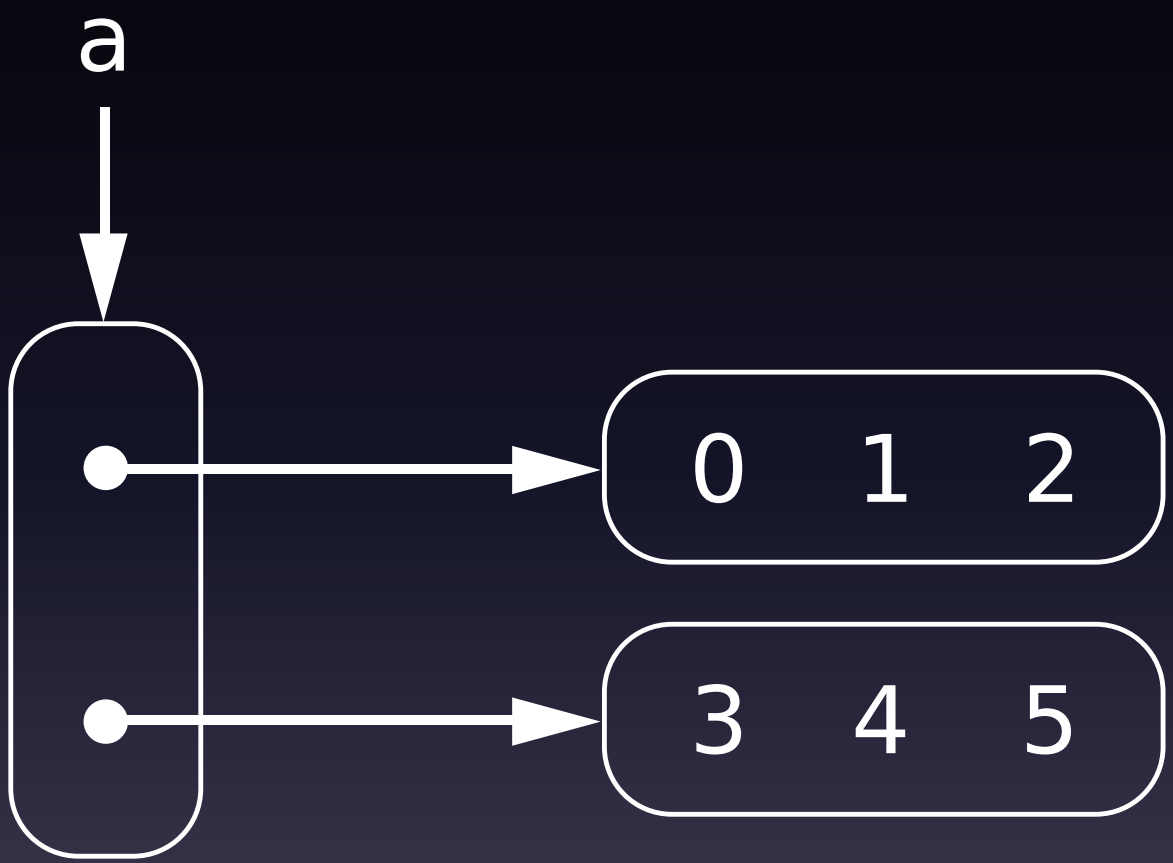
2D-Array als Array von Pointern



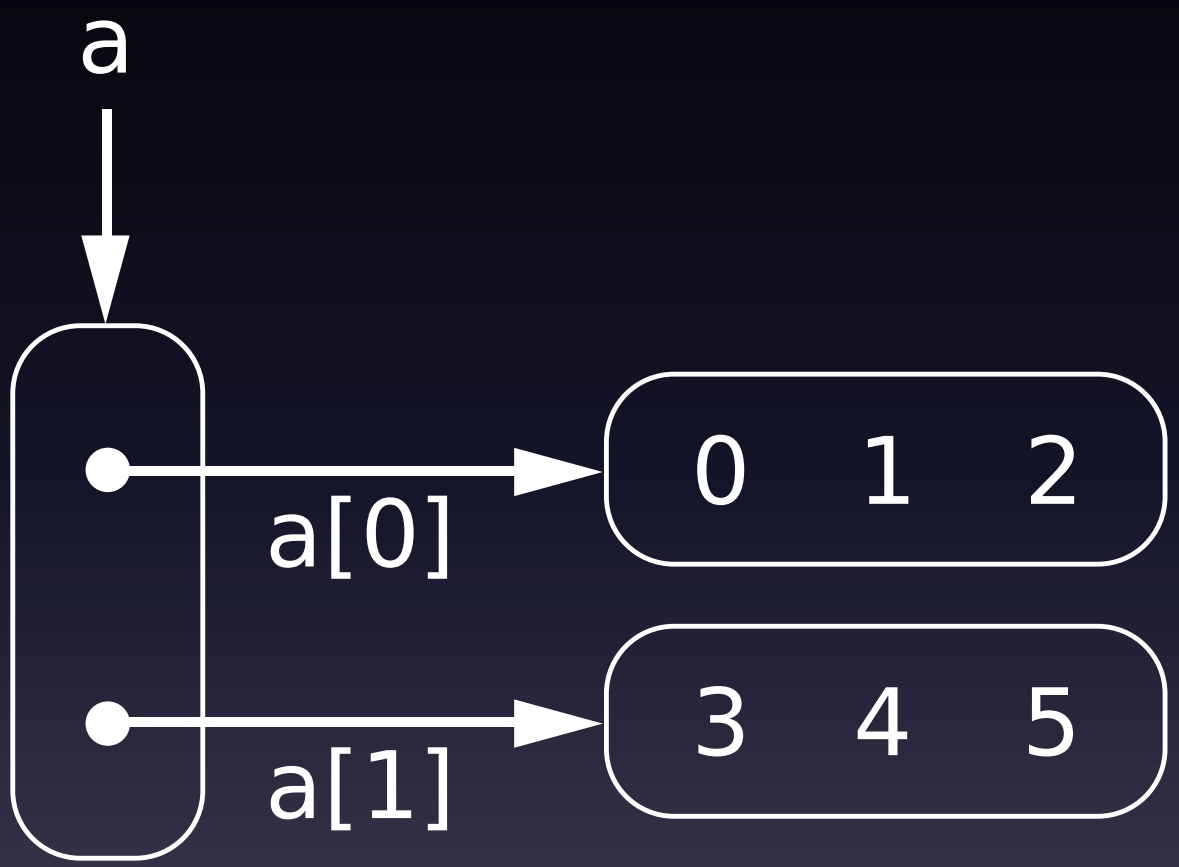
2D-Array als Array von Pointern



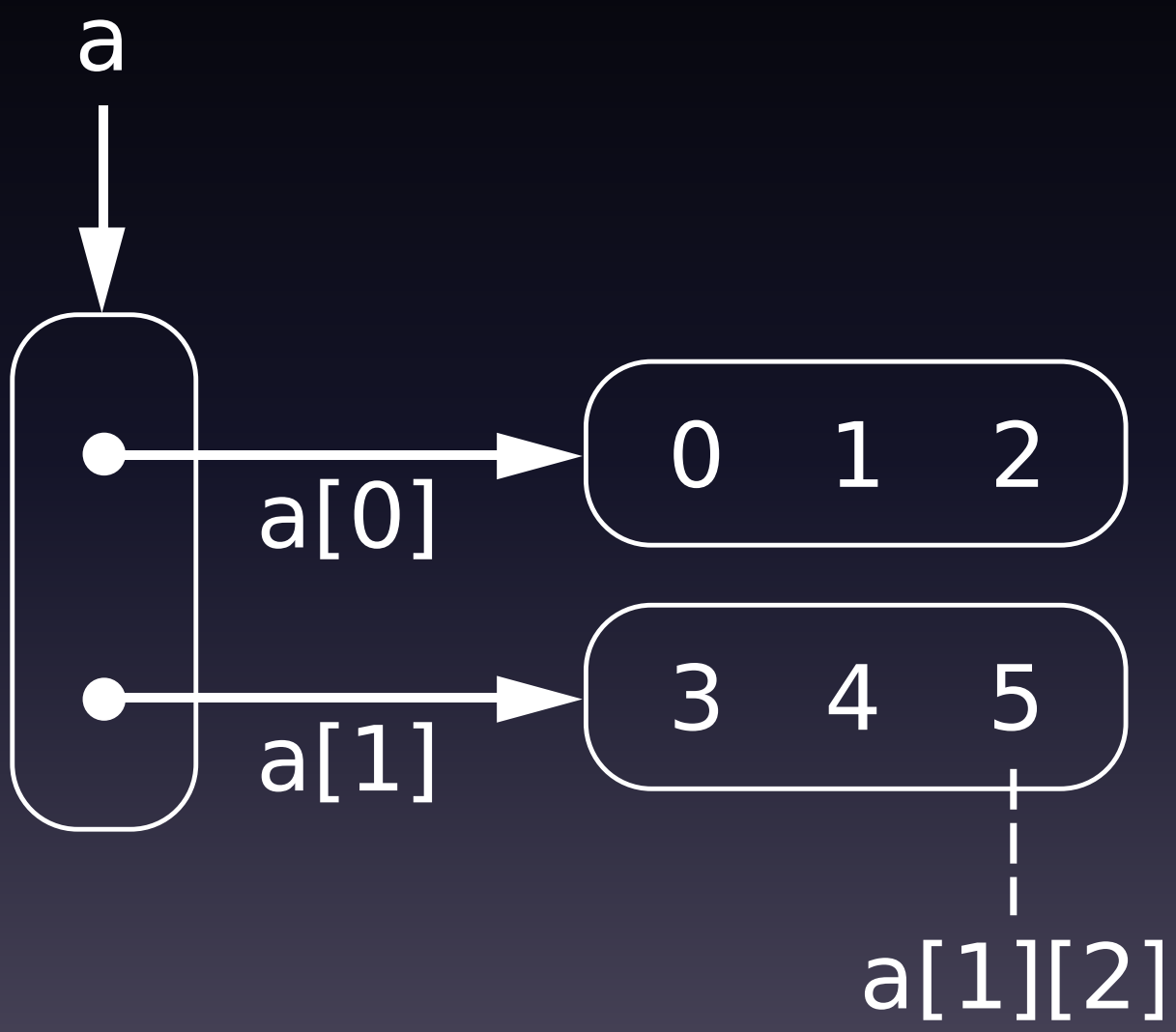
2D-Array als Array von Pointern



2D-Array als Array von Pointern



2D-Array als Array von Pointern



2D-Array als Array von Pointern

```
int main() {




  return 0;
}
```

2D-Array als Array von Pointern

```
int main() {
  int row0[] = {0, 1, 2};
  int row1[] = {3, 4, 5};
  int * const d[] = {row0, row1};

  return 0;
}
```

2D-Array als Array von Pointern

```
int main() {
  int row0[] = {0, 1, 2};
  int row1[] = {3, 4, 5};
  int * const d[] = {row0, row1};
  demo_deep(d);
  return 0;
}
```

2D-Array als Array von Pointern

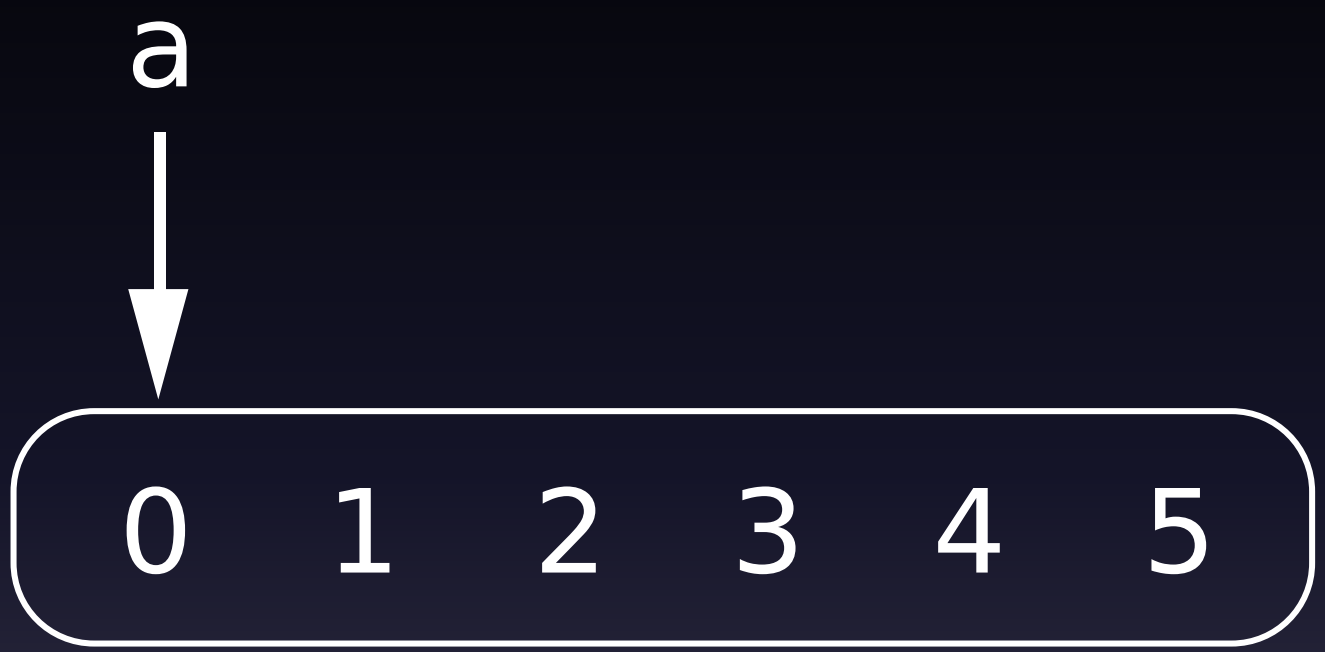
```
void demo_deep(int * const *a) {
  a[1][2] *= 2;
}

int main() {
  int row0[] = {0, 1, 2};
  int row1[] = {3, 4, 5};
  int * const d[] = {row0, row1};
  demo_deep(d);
  return 0;
}
```

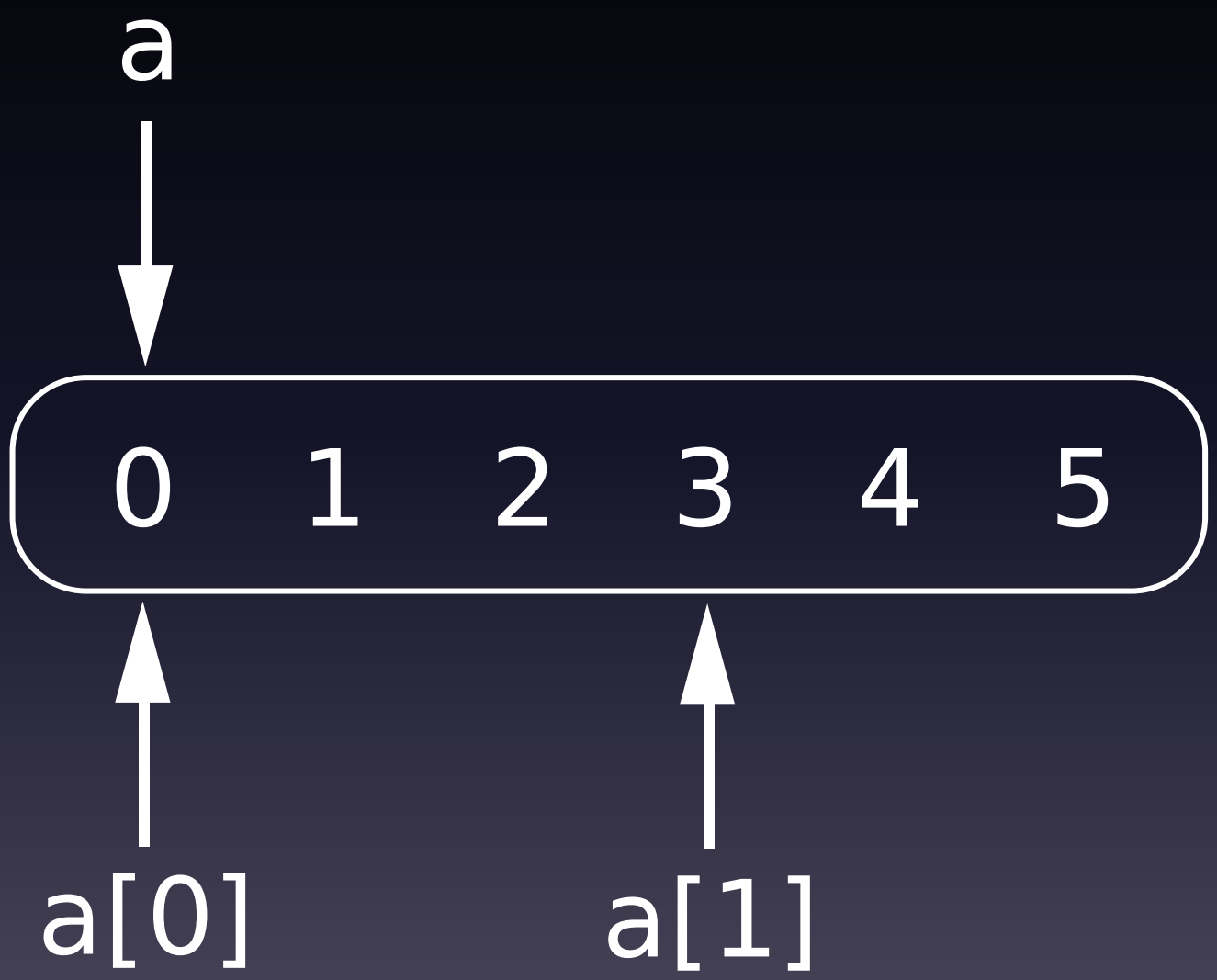
Variante "Linear"

2D-Array linear

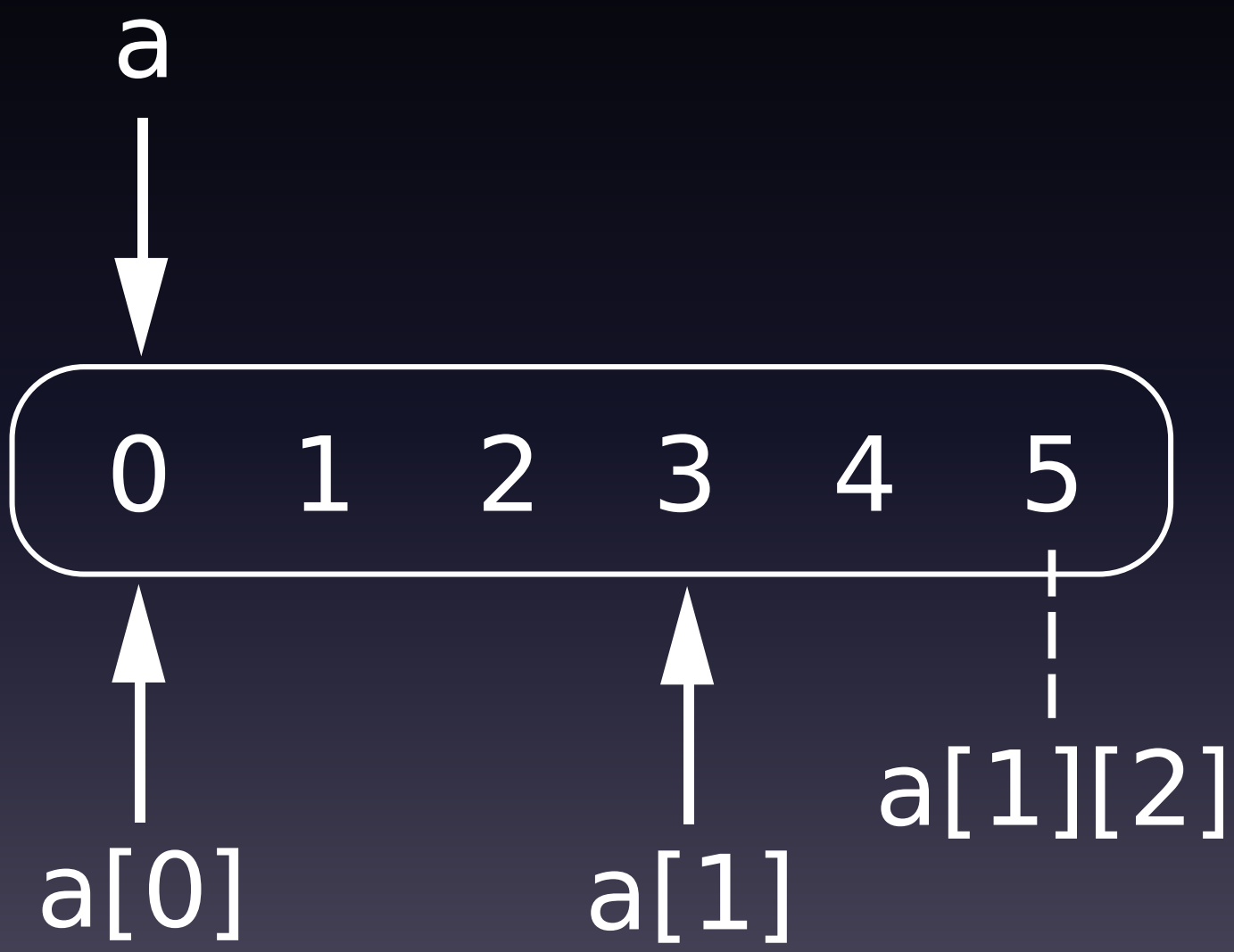
2D-Array linear



2D-Array linear



2D-Array linear



2D-Array linear

```
int main() {


  return 0;
}
```

2D-Array linear

```
int main() {
  int f[][3] = { {0, 1, 2},
                 {3, 4, 5} };

  return 0;
}
```

2D-Array linear

```
int main() {
  int f[][3] = { {0, 1, 2},
                 {3, 4, 5} };
  demo_flat(f);
  return 0;
}
```

2D-Array linear

```
void demo_flat(int a[][3]) {
  a[1][2] *= 2;
}

int main() {
  int f[][3] = { {0, 1, 2},
                 {3, 4, 5} };
  demo_flat(f);
  return 0;
}
```

Variante "Hybrid"

2D-Array-Hybrid

```
int main() {




  return 0;
}
```

2D-Array-Hybrid

```
int main() {
  int f[][3] = { {0, 1, 2},
                 {3, 4, 5} };



  return 0;
}
```

2D-Array-Hybrid

```
int main() {
  int f[][3] = { {0, 1, 2},
                 {3, 4, 5} };
  int * const d[] = {f[0], f[1]};


  return 0;
}
```

2D-Array-Hybrid

```
int main() {
  int f[][3] = { {0, 1, 2},
                 {3, 4, 5} };
  int * const d[] = {f[0], f[1]};
  demo_flat(f);
  demo_deep(d);
  return 0;
}
```

2D-Array-Hybrid

```
void demo_flat(int a[][3]);
void demo_deep(int * const *a);

int main() {
  int f[][3] = { {0, 1, 2},
                 {3, 4, 5} };
  int * const d[] = {f[0], f[1]};
  demo_flat(f);
  demo_deep(d);
  return 0;
}
```

Deklaration von Strings

Deklaration von Strings

```
char a[] = "Hallo";
```

Deklaration von Strings

```
char a[] = "Hallo";
sizeof(a) == sizeof(char)*(5 + 1)
```

Deklaration von Strings

Variante "Mit Array"

```
char a[] = "Hallo";
sizeof(a) == sizeof(char)*(5 + 1)
```

Deklaration von Strings

Variante "Mit Array"

```
char a[] = "Hallo";
sizeof(a) == sizeof(char)*(5 + 1)
```

Inhalt les- und schreibbar

Deklaration von Strings

Variante "Mit Array"

```
char a[] = "Hallo";
sizeof(a) == sizeof(char)*(5 + 1)
```

Inhalt les- und schreibbar

Variante "Nur Pointer"

Deklaration von Strings

Variante "Mit Array"

```
char a[] = "Hallo";
sizeof(a) == sizeof(char)*(5 + 1)
```

Inhalt les- und schreibbar

Variante "Nur Pointer"

```
char *p = "Hallo";
```

Deklaration von Strings

Variante "Mit Array"

```
char a[] = "Hallo";
sizeof(a) == sizeof(char)*(5 + 1)
```

Inhalt les- und schreibbar

Variante "Nur Pointer"

```
char *p = "Hallo";
sizeof(p) == sizeof(char *)
```

Deklaration von Strings

Variante "Mit Array"

```
char a[] = "Hallo";
sizeof(a) == sizeof(char)*(5 + 1)
```

Inhalt les- und schreibbar

Variante "Nur Pointer"

```
char *p = "Hallo";
sizeof(p) == sizeof(char *)
```

Inhalt *nicht* schreibbar

Deklaration von Strings

Variante "Mit Array"

```
char a[] = "Hallo";
sizeof(a) == sizeof(char)*(5 + 1)
```

Inhalt les- und schreibbar

Variante "Nur Pointer"

```
const char *p = "Hallo";
sizeof(p) == sizeof(char *)
```

Inhalt *nicht* schreibbar

NULL-Pointer

NULL-Pointer

```
#include <stdlib.h>  /* for NULL */
...
int *p = NULL;
```

NULL-Pointer

```
#include <stdlib.h>  /* for NULL */
...
int *p = NULL;
...
int b = *p;
```

NULL-Pointer

```
#include <stdlib.h>  /* for NULL */
...
int *p = NULL;
...
int b = *p; ← BANG!
```

Weiterführende Themen

Weiterführende Themen

- Character encodings, Unicode, wchar_t
- Sicherer Umgang mit Strings
- Der Typ size_t
- Void-Pointer
- Funktions-Pointer

Zusammenfassung

Zusammenfassung

- `sizeof(`x`)` liefert die Größe von x in Byte
- &-Operator liefert eine Adresse
- *-Operator dereferenziert
- Pointer sind typisiert
- Arrays sind auch Pointer auf ihr erstes Element
- Strings sind nullterminiiniert
- const correctness bei Pointern ist wichtig

Danke!

Fragen?

http://blog.hartwork.org/