C-Kurs 2010
Pointer

Sebastian@Pipping.org

16. September 2010

v2.7

C-Kurs

	Mi	Konzepte, Syntax, ... printf, scanf
Next →	Do	Pointer, Arrays, ... Compiler, Headers, ...
	Fr	Structs, malloc, ... Debugging I, ... + Vorstellung der Aufgabe
	Mo	Lösung der Aufgabe
	Di	Debugging II stdlib, Bücher, ...

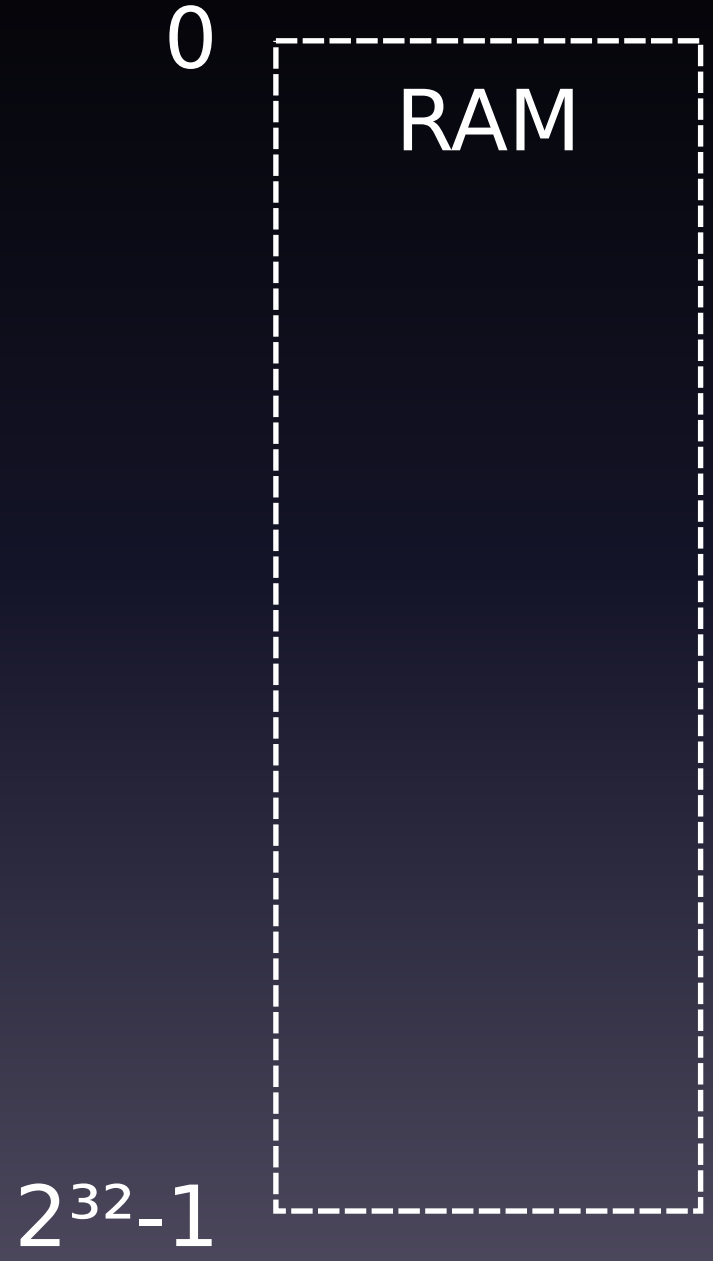
Ohne Pointer geht nichts.

Themen

- Hello Pointer
- Call by Reference
- `NULL`-Pointer
- Pointer auf Pointer
- Von Pointern zu Arrays
- Pointer-Arithmetik
- Strings und Längenberechnung
- Const Correctness
- Initialisierung von Strings
- Mehrdimensionale Arrays
- Programm-Argumente: `argc` und `argv`
- Weiterführende Themen
- Zusammenfassung

Pointer

Variablen im RAM

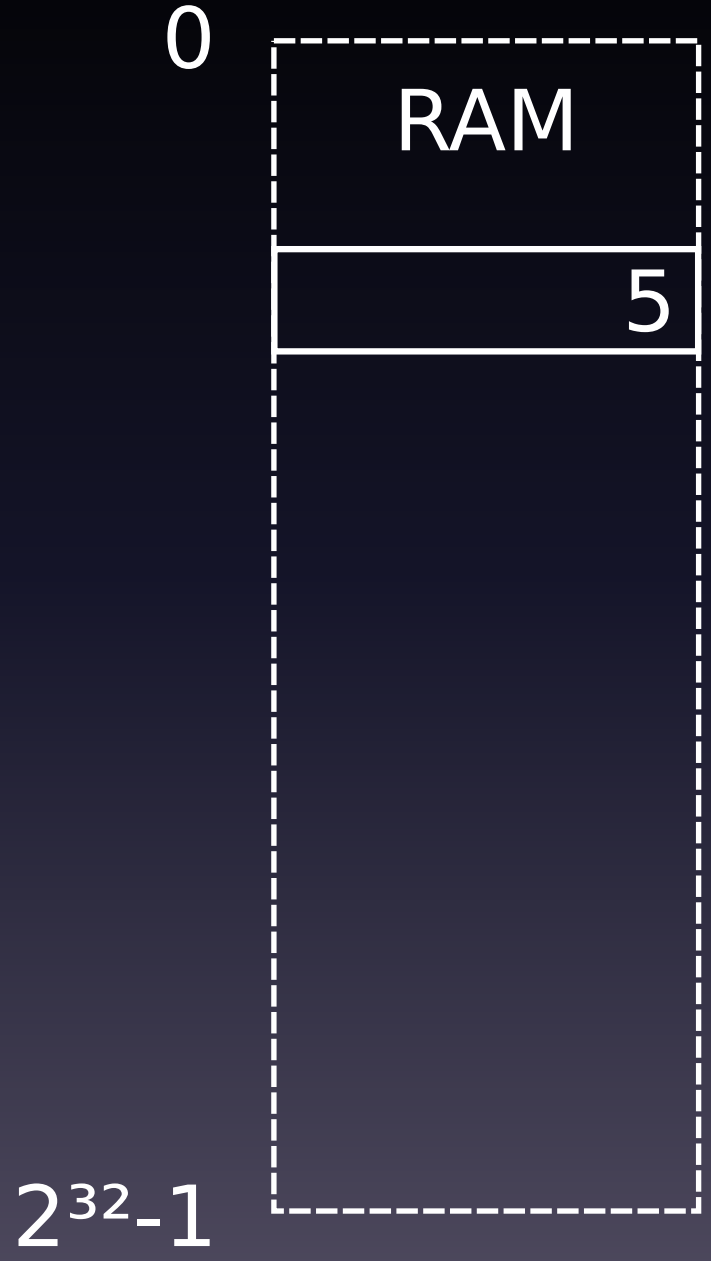


Variablen im RAM

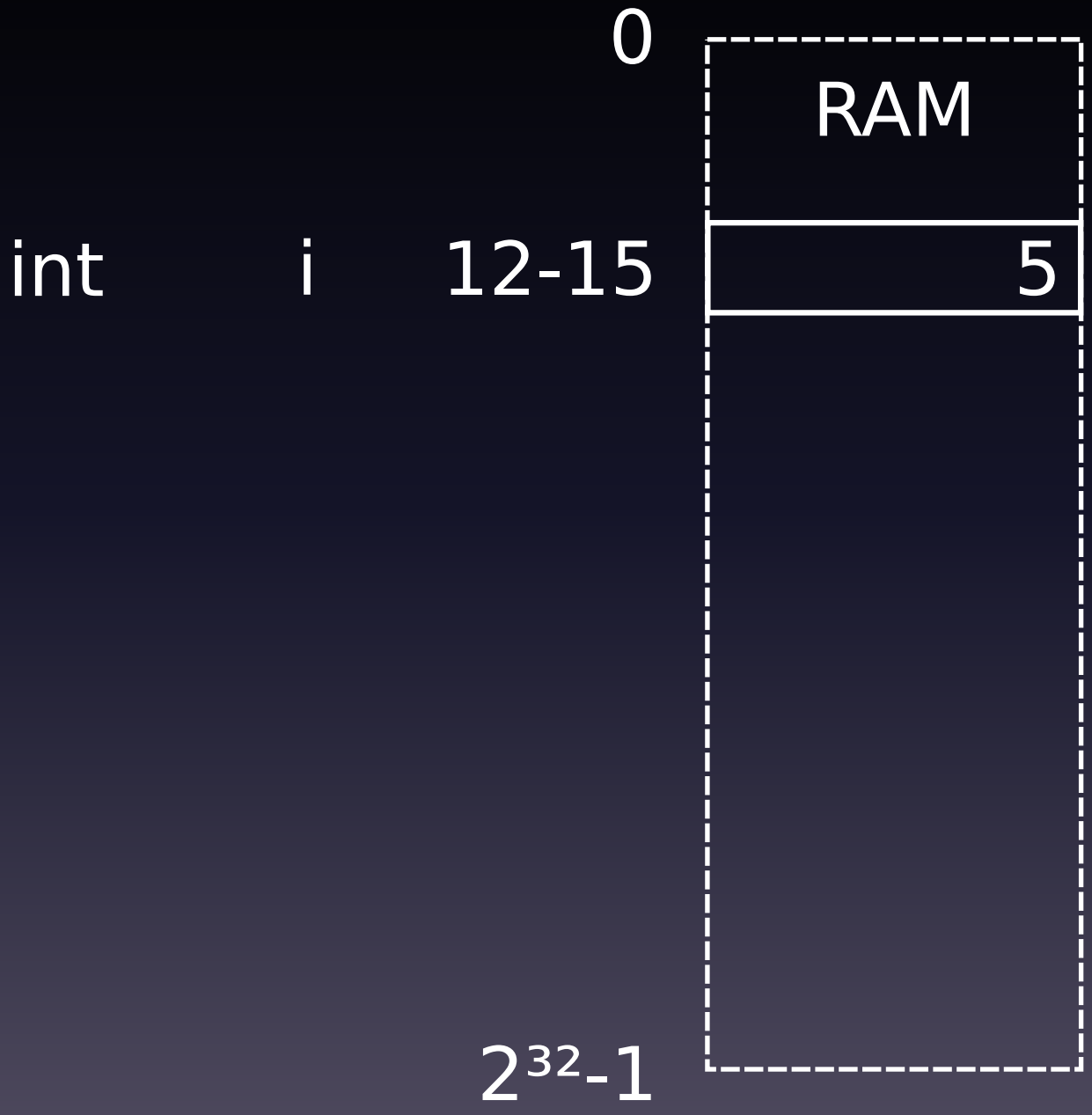
int i

Variablen im RAM

int i

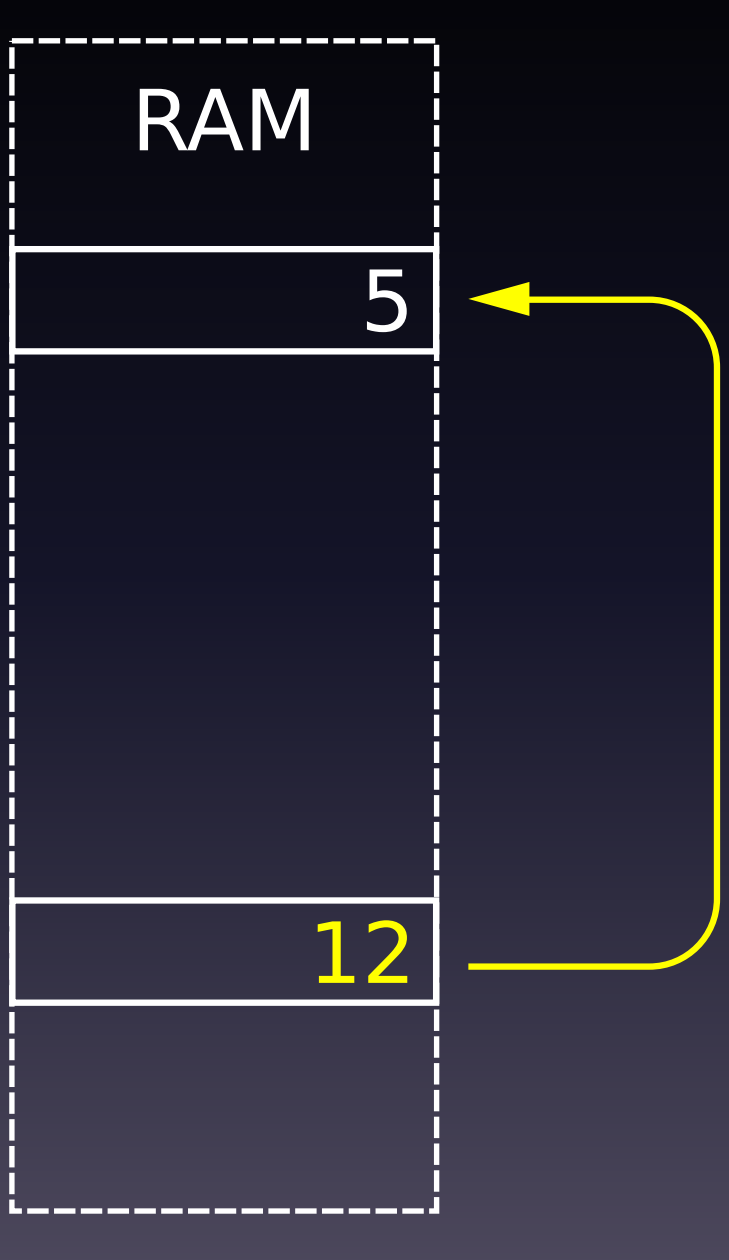


Variablen im RAM

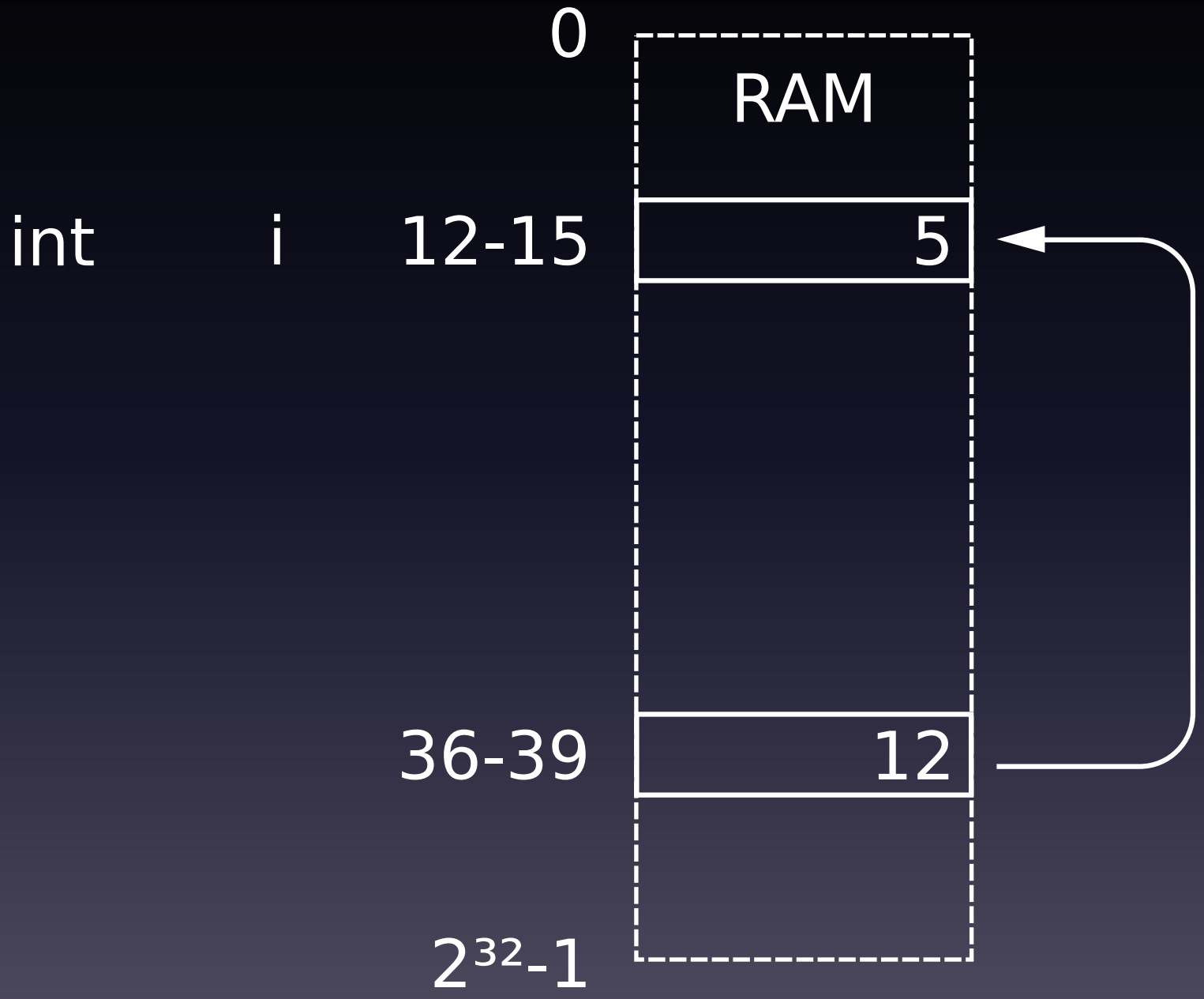


Variablen im RAM

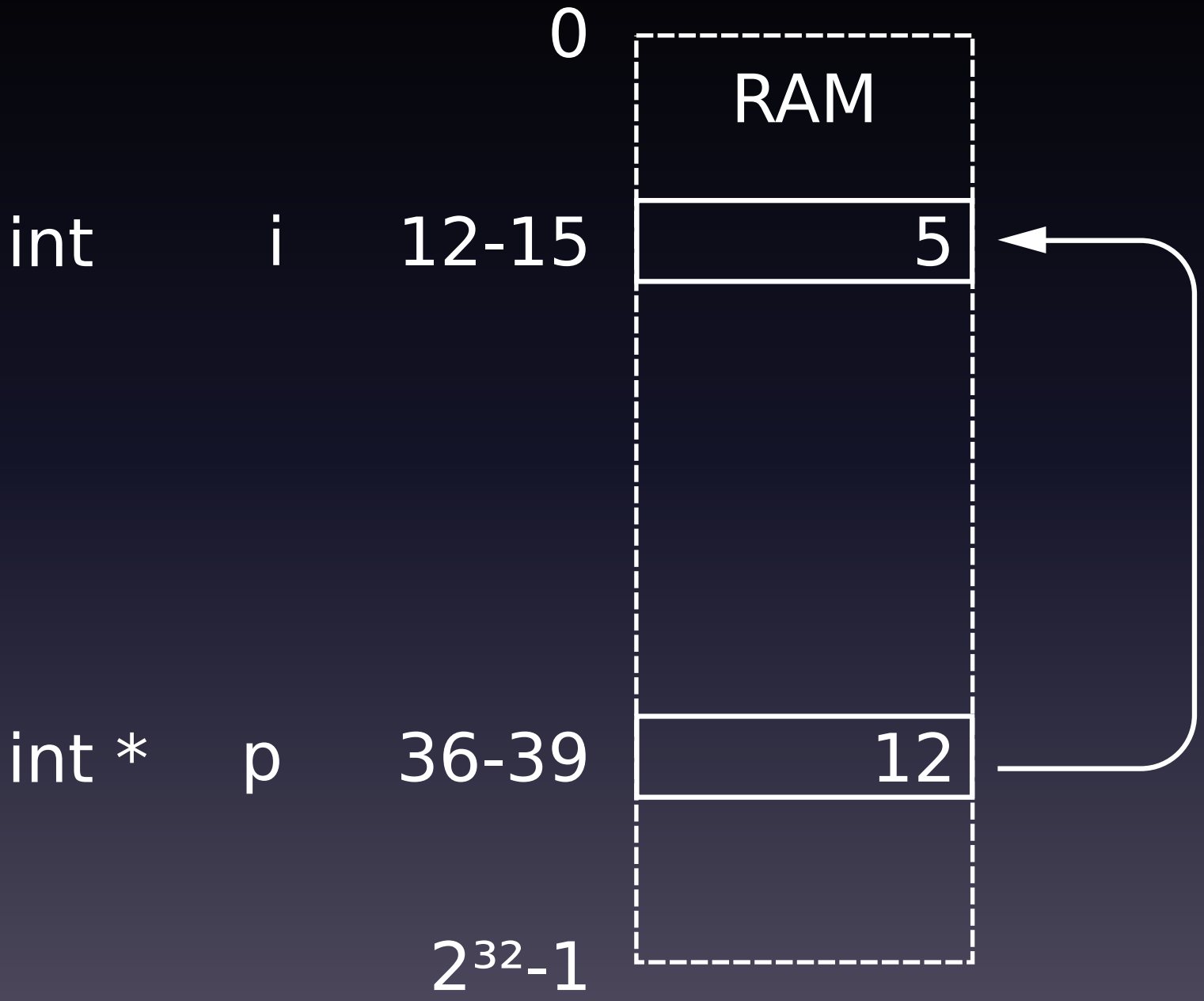
int i 12-15



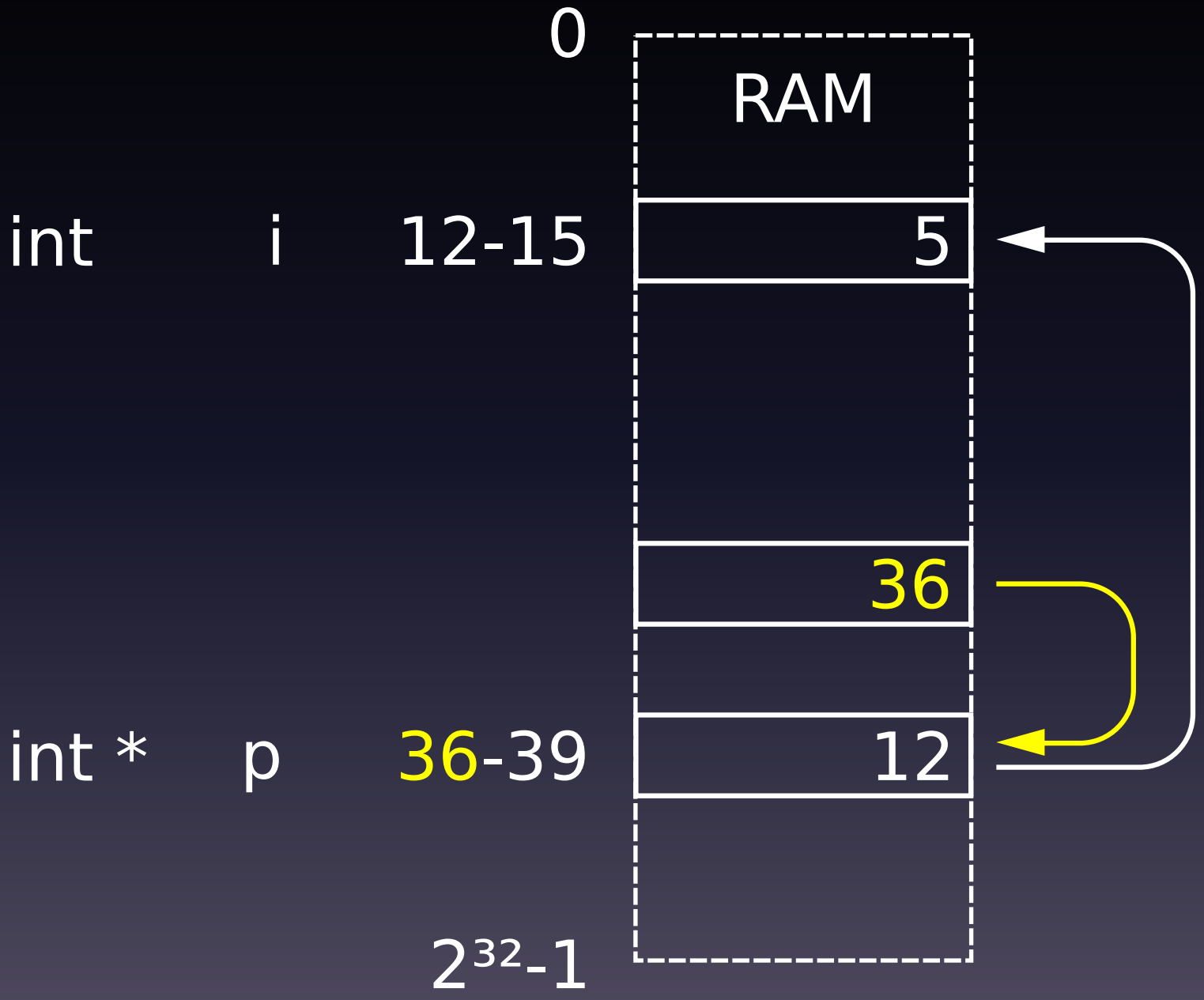
Variablen im RAM



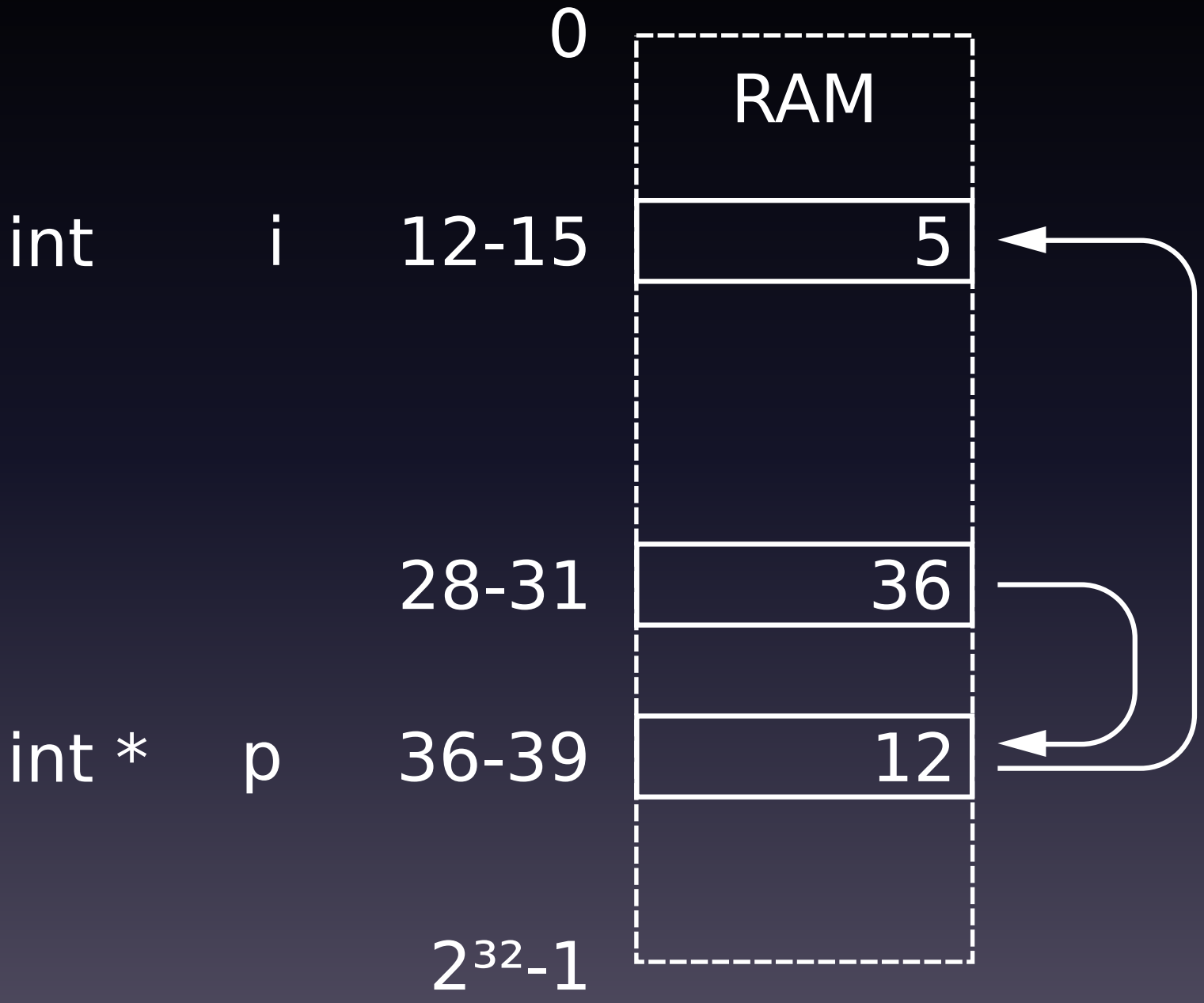
Variablen im RAM



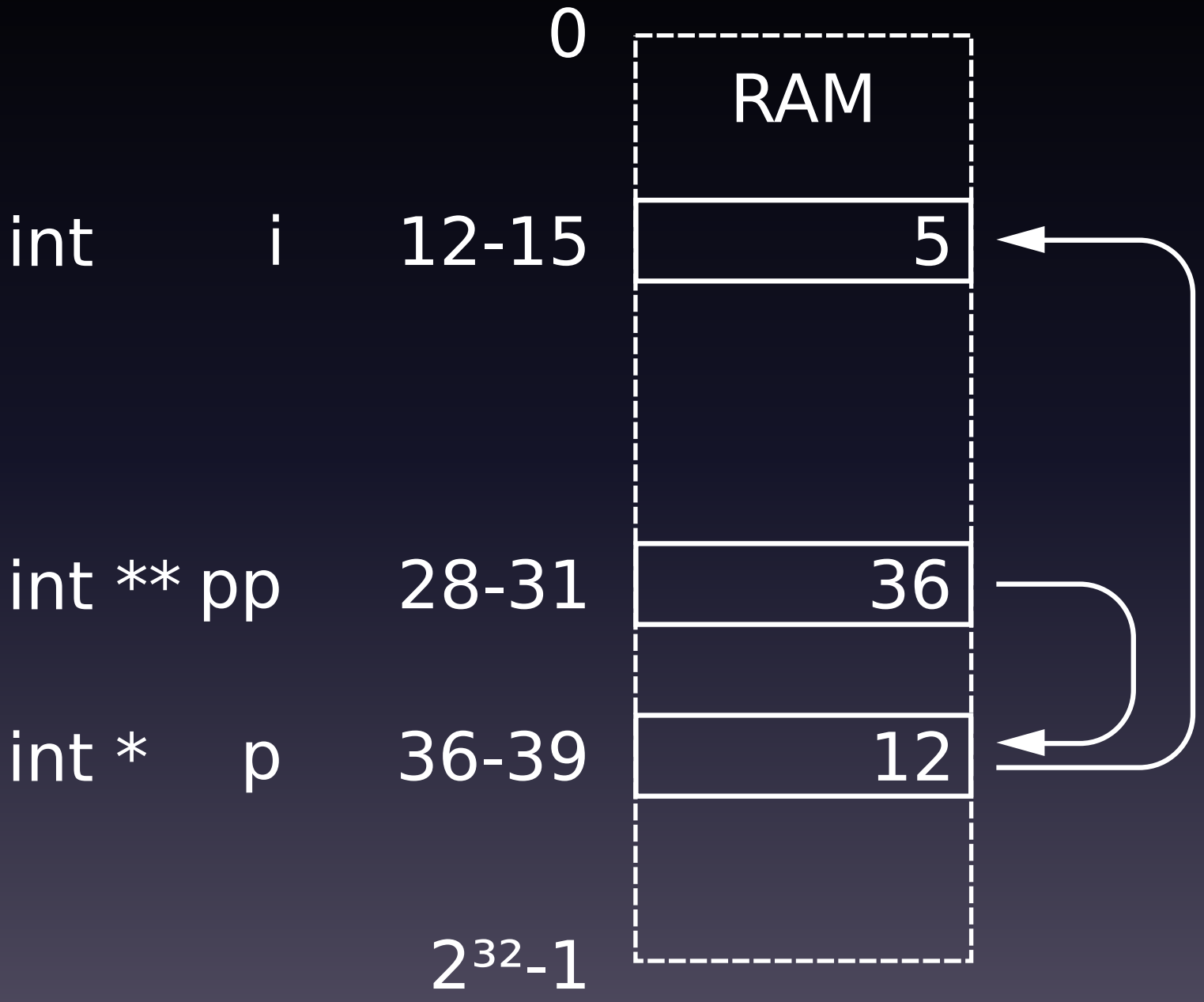
Variablen im RAM



Variablen im RAM



Variablen im RAM



Was ist ein „Pointer“?

„Pointer“ kann meinen eine...

A)

B)

„Pointer“ kann meinen eine...

A) Adresse
B)

„Pointer“ kann meinen eine...

A) Adresse
B) Variable, die eine Adresse speichert

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;


  *p = 4;




  return 0;
}
```

Hello Pointer

```
#include <stdio.h>

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


  *p = 4;

  p = 5;


  return 0;
}
```

Hello Pointer

```
#include <stdio.h>

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

  printf(“i == %d, p == %p\n”, i, p);
  *p = 4;
  printf(“i == %d, p == %p\n”, i, p);
  p = 5;
  printf(“i == %d, p == %p\n”, i, p);

  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
i == 3, p == 0xbfa3ad8c
```

Ausgabe

```
# gcc hello_pointer.c -o hello_pointer
# ./hello_pointer
i == 3, p == 0xbfa3ad8c
                        | *p = 4;
```

Ausgabe

```
# gcc hello_pointer.c -o hello_pointer
# ./hello_pointer
i == 3, p == 0xbfa3ad8c
                        | *p = 4;
i == 4, p == 0xbfa3ad8c
```

Ausgabe

```
# gcc hello_pointer.c -o hello_pointer
# ./hello_pointer
i == 3, p == 0xbfa3ad8c
                          | *p = 4;
i == 4, p == 0xbfa3ad8c
                          | p = 5;
```

Ausgabe

```
# gcc hello_pointer.c -o hello_pointer
# ./hello_pointer
i == 3, p == 0xbfa3ad8c
                          | *p = 4;
i == 4, p == 0xbfa3ad8c
                          | p = 5;
i == 4, p == 0x5
```

Zusammenfassung 1/2

- Operator `&` liefert eine Adresse
-
-

Zusammenfassung 1/2

- Operator `&` liefert eine Adresse
- Operator $*$ folgt einer Adresse
  (er *dereferenziert*)
-

Zusammenfassung 1/2

- Operator `&` liefert eine Adresse
- Operator `*` folgt einer Adresse
  (er *dereferenziert*)
- `&` und `*` sind komplementär; es gilt:

```
*(&x) = x
```

Zusammenfassung 2/2

Pointer zeigen auf typisierte Daten:

`int *` $\neq$ `char *`

Call by Reference

Call by Reference

```
#include <stdio.h>




int main() {
  int i = 0;









  return 0;
}
```

Call by Reference

```
#include <stdio.h>




int main() {
  int i = 0;

  by_value(i);




  return 0;
}
```

Call by Reference

```
#include <stdio.h>




int main() {
  int i = 0;

  by_value(i);

  by_reference(&i);


  return 0;
}
```

Call by Reference

```
#include <stdio.h>

void by_value    (int   x) {   x += 3;  }
void by_reference(int * x) {  *x += 4;  }

int main() {
  int i = 0;

  by_value(i);

  by_reference(&i);


  return 0;
}
```

Call by Reference

```
#include <stdio.h>

void by_value    (int   x) {   x += 3;  }
void by_reference(int * x) {  *x += 4;  }

int main() {
  int i = 0;

  by_value(i);
  printf("i == %d\n", i);
  by_reference(&i);
  printf("i == %d\n", i);

  return 0;
}
```

Ausgabe

```
# gcc  call_by_reference.c \
       -o call_by_reference
```

Ausgabe

```
# gcc  call_by_reference.c \
       -o call_by_reference
# ./call_by_reference
```

Ausgabe

```
# gcc  call_by_reference.c \
       -o call_by_reference
# ./call_by_reference
i == 0
i == 4
```

`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!
```

Zusammenfassung

- `NULL` markiert Abwesenheit

Zusammenfassung

- `NULL` markiert Abwesenheit
- `NULL`-Pointern darf nicht gefolgt werden

Zusammenfassung

- `NULL` markiert Abwesenheit
- `NULL`-Pointern darf nicht gefolgt werden
- Wir müssen wissen, ob ein Pointer `NULL` ist, bevor wir ihm folgen

Pointer auf Pointer

Anwendungen

- Mehrdimensionale Arrays
-

Anwendungen

- Mehrdimensionale Arrays
- Call by Reference *von* Pointern

man strtod

```
STRTOD(3)    Linux Programmer's Manual    STRTOD(3)

NAME
   strtod, strtof, strtold - convert ASCII string
                            to floating-point number

SYNOPSIS
   #include <stdlib.h>

   double strtod(const char *nptr, char **endptr);
   float strtof(const char *nptr, char **endptr);

...
```

man strtod

```
STRTOD(3)    Linux Programmer's Manual    STRTOD(3)

NAME
   strtod, strtof, strtold - convert ASCII string
                            to floating-point number

SYNOPSIS
   #include <stdlib.h>

   double strtod(const char *nptr, char **endptr);
   float strtof(const char *nptr, char **endptr);

...
```

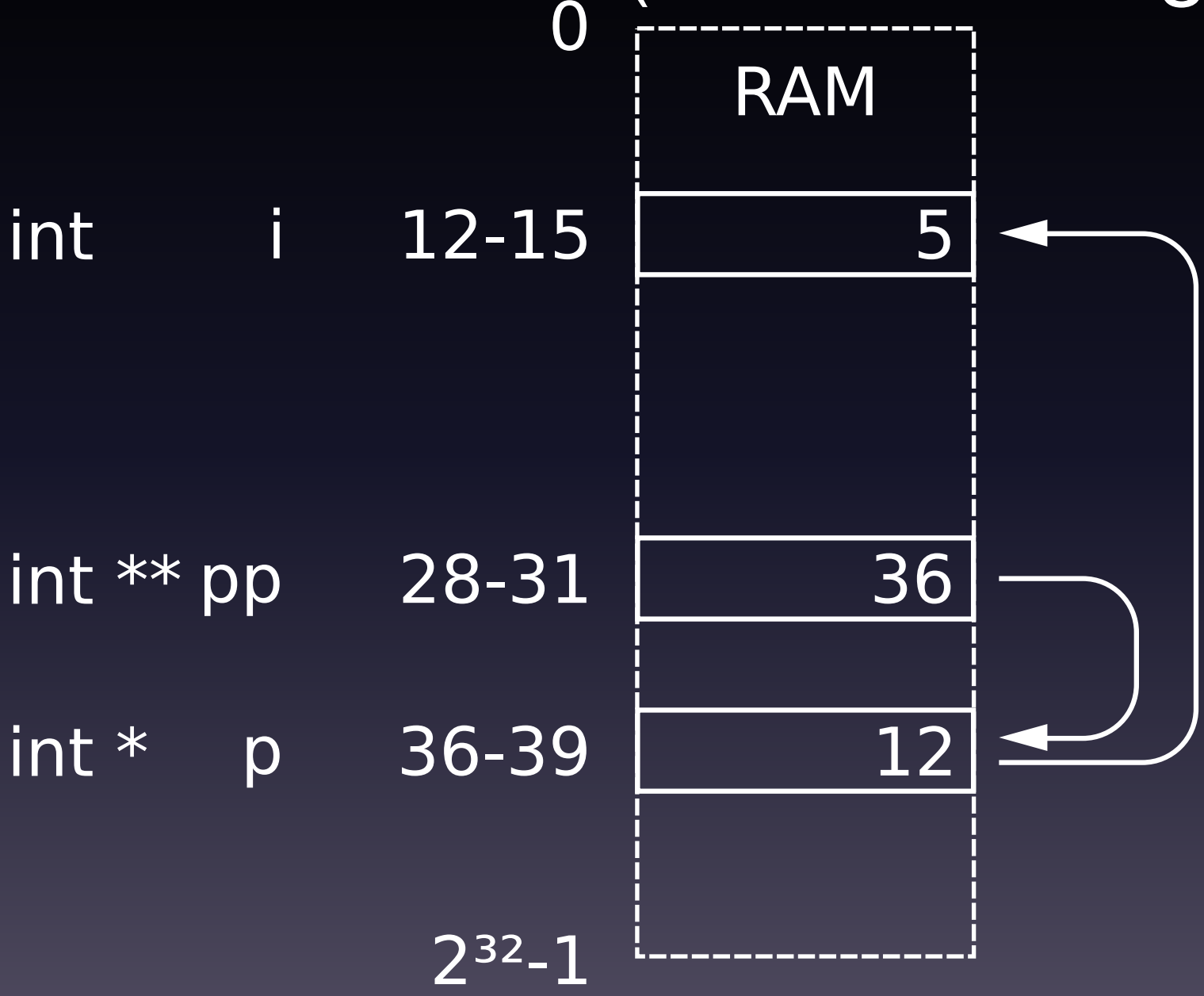
Pointer auf Pointer

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

Pointer auf Pointer

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

Variablen im RAM (Wiederholung)



Von Pointern zu Arrays

Von Pointern zu Arrays

```
#include <stdio.h>

int main() {

  return 0;
}
```

Von Pointern zu Arrays

```
#include <stdio.h>

void dump(const int * data, int count) {




}

int main() {


  return 0;
}
```

Von Pointern zu Arrays

```
#include <stdio.h>

void dump(const int * data, int count) {
  int i = 0;
  for (; i < count; ++i) {

  }
}

int main() {


  return 0;
}
```

Von Pointern zu Arrays

```
#include <stdio.h>

void dump(const int * data, int count) {
  int i = 0;
  for (; i < count; ++i) {
    printf("Field %d: %d\n", i + 1, *(data + i));
  }
}

int main() {


  return 0;
}
```

Von Pointern zu Arrays

```
#include <stdio.h>

void dump(const int * data, int count) {
  int i = 0;
  for (; i < count; ++i) {
    printf(“Field %d: %d\n”, i + 1, data[i]);
  }
}

int main() {


  return 0;
}
```

Von Pointern zu Arrays

```
#include <stdio.h>

void dump(const int * data, int count) {
  int i = 0;
  for (; i < count; ++i) {
    printf(“Field %d: %d\n”, i + 1, data[i]);
  }
}

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

  return 0;
}
```

Von Pointern zu Arrays

```
#include <stdio.h>

void dump(const int * data, int count) {
  int i = 0;
  for (; i < count; ++i) {
    printf(“Field %d: %d\n”, i + 1, data[i]);
  }
}

int main() {
  int const primes[] = {2, 3, 5, 7, 11};
  dump(primes, 5);
  return 0;
}
```

Von Pointern zu Arrays

```
#include <stdio.h>

void dump(const int * data, int count) {
  int i = 0;
  for (; i < count; ++i) {
    printf("Field %d: %d\n", i + 1, data[i]);
  }
}

int main() {
  int const primes[] = {2, 3, 5, 7, 11};
  dump(primes, sizeof(primes) / sizeof(int));
  return 0;
}
```

Von Pointern zu Arrays

```
#include <stdio.h>

void dump(const int * data, int count) {
  int i = 0;
  for (; i < count; ++i) {
    printf("Field %d: %d\n", i + 1, data[i]);
  }
}

int main() {
  int const primes[] = {2, 3, 5, 7, 11, 13};
  dump(primes, sizeof(primes) / sizeof(int));
  return 0;
}
```

Zusammenfassung (1/5)

Array-Zugriff via `[`n`]` ähnlich Java

Zusammenfassung (2/5)

Array = Pointer auf das erste Element

Zusammenfassung (3/5)

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

Zusammenfassung (4/5)

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

=

Von *`variable`* belegter Speicher in Byte

Zusammenfassung (5/5)

`sizeof(`*type*`)`

=

Von *type*-Instanzen belegter Speicher in Byte

Pointer-Arithmetik

Pointer-Arithmetik

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

Pointer-Arithmetik

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

Es gilt:

1. `numbers[i] = *(numbers + i)`
2.
3.

Pointer-Arithmetik

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

Es gilt:

1. `numbers[i] = *(numbers + i)`
2. `text[j] = *(text + j)`
3.

Pointer-Arithmetik

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

Es gilt:

1. `numbers[i] = *(numbers + i)`
2. `text[j] = *(text + j)`
3. `sizeof(char)` $\neq$ `sizeof(int)`

Pointer-Arithmetik

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

Es gilt:

1. `numbers[i] = *(numbers + i)`
2. `text[j] = *(text + j)`
3. `sizeof(char)` $\neq$ `sizeof(int)`

Für Operatoren +/- auf Pointern folgt:

Pointer-Arithmetik

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

Es gilt:

1. `numbers[i] = *(numbers + i)`
2. `text[j] = *(text + j)`
3. `sizeof(char)` $\neq$ `sizeof(int)`

Für Operatoren +/- auf Pointern folgt:
Sprungweite variiert mit dem Typen!

Strings

Strings

“ABC”

Strings

```
“ABC” = { 65, 66, 67, 0 }
```

Strings

```
“ABC” = { 65, 66, 67, 0 }
“012”
```

Strings

```
“ABC” = { 65, 66, 67, 0 }
“012” = { 48, 49, 50, 0 }
```

Strings

```
“ABC” = { 65, 66, 67, 0 }
“012” = { 48, 49, 50, 0 }
“012\0”
```

Strings

```
“ABC” = { 65, 66, 67, 0 }
“012” = { 48, 49, 50, 0 }
“012\0” = { 48, 49, 50, 0, 0 }
```

Strings

```
“ABC” = { 65, 66, 67, 0 }
“012” = { 48, 49, 50, 0 }
“012\0” = { 48, 49, 50, 0, 0 }

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

Strings

```
“ABC” = { 65, 66, 67, 0 }
“012” = { 48, 49, 50, 0 }
“012\0” = { 48, 49, 50, 0, 0 }

#include <string.h>
...
strlen(“ABC”);
= 3
```

my_strlen

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




  return (                );
}
```

my_strlen

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



  return (                );
}
```

my_strlen

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

my_strlen

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

Const Correctness

Const Correctness (1/6)

Modifikator `const`
verbietet Schreibzugriff

Const Correctness (2/6)

```
_____ int _____ foo;
```

Const Correctness (2/6)

```
const int _____ foo;
```

Const Correctness (2/6)

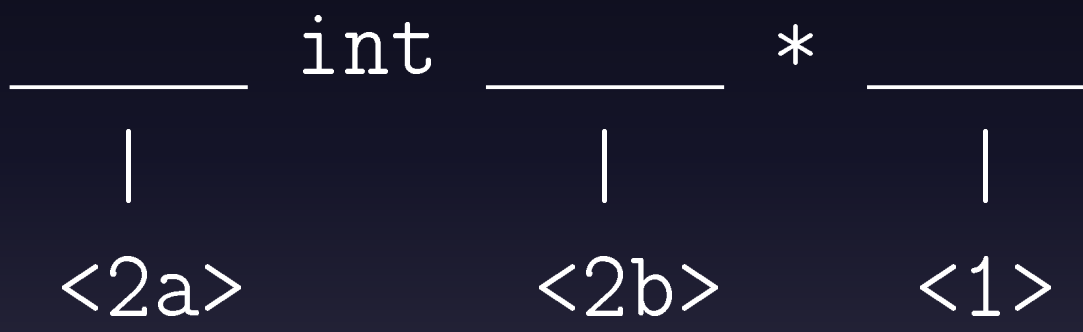
```
_____ int const foo;
```

Const Correctness (3/6)

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

Const Correctness (3/6)

Const Correctness (3/6)



Const Correctness (4/6)

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

Const Correctness (4/6)

<1> _____ int _____ * const foo;

Verboten:

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

Const Correctness (4/6)

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

Verboten:

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

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

Const Correctness (4/6)

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

Verboten:

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

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

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

Verboten:

```
foo[0] = ...;
```

Const Correctness (5/6)

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

Const Correctness (5/6)

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

Verboten:

```
   foo = ...;
foo[0] = ...;
```

Const Correctness (6/6)

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

Const Correctness (6/6)

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

Erlaubt:

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

Const Correctness (6/6)

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

Erlaubt:

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

Verboten:

```
foo[0] = ...;
foo[0][0] = ...;
```

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

Initialisierung von Strings

Initialisierung von Strings

```
char a[] = “Hallo”;
```

Initialisierung von Strings

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

Initialisierung von Strings

Variante „Mit Array“

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

Initialisierung von Strings

Variante „Mit Array“

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

Inhalt les- und schreibbar

Initialisierung von Strings

Variante „Mit Array“

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

Inhalt les- und schreibbar

Variante „Nur Pointer“

Initialisierung von Strings

Variante „Mit Array“

```
char a[] = ‘‘Hallo’’;
```

```
sizeof(a) = sizeof(char)*(5 + 1)
```

Inhalt les- und schreibbar

Variante „Nur Pointer“

```
char * p = ‘‘Hallo’’;
```

Initialisierung 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 *)
```

Initialisierung 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

Initialisierung 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

Variante „Nur Pointer“

```
#include <stdio.h>

int main() {
  char * a = “hallo”;
  char * b = “hallo”;









  return 0;
}
```

Variante „Nur Pointer“

```
#include <stdio.h>

int main() {
  char * a = “hallo”;
  char * b = “hallo”;

  printf(“a=%p\n”
         “b=%p\n”, a, b);



  return 0;
}
```

Variante „Nur Pointer“

```
#include <stdio.h>

int main() {
  char * a = “hallo”;
  char * b = “hallo”;

  printf(“a=%p\n”
         “b=%p\n”, a, b);

  a[0] = ‘X’;

  return 0;
}
```

Ausgabe

```
# gcc rombad.c -o rombad
```

Ausgabe

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

Ausgabe

```
# gcc rombad.c -o rombad
# ./rombad
a=0x80484dc
b=0x80484dc
```

Ausgabe

```
# gcc rombad.c -o rombad
# ./rombad
a=0x80484dc
b=0x80484dc
```

Ausgabe

```
# gcc rombad.c -o rombad
# ./rombad
a=0x80484dc
b=0x80484dc
Segmentation fault
```

GCC Flag -Wwrite-strings

```
# gcc  -Wall -Wextra -Wwrite-strings \
       rombad.c  -o rombad
```

GCC Flag -Wwrite-strings

```
# gcc  -Wall -Wextra -Wwrite-strings \
      rombad.c  -o rombad
rombad.c: In function ‘main’:
rombad.c:4: warning: initialization discards
    qualifiers from pointer target type
rombad.c:5: warning: initialization discards
    qualifiers from pointer target type
```

Variante „Nur Pointer“

```
#include <stdio.h>

int main() {
  char * a = “hallo”;
  char * b = “hallo”;

  printf(“a=%p\n”
         “b=%p\n”, a, b);

  a[0] = ‘X’;

  return 0;
}
```

Variante „Nur Pointer“

```
#include <stdio.h>

int main() {
  const char * a = “hallo”;
  const char * b = “hallo”;

  printf(“a=%p\n”
         “b=%p\n”, a, b);

  a[0] = ‘X’;

  return 0;
}
```

Variante „Nur Pointer“

```
#include <stdio.h>

int main() {
  const char * a = “hallo”;
  const char * b = “hallo”;

  printf(“a=%p\n”
         “b=%p\n”, a, b);

  a[0] = ‘X’;

  return 0;
}
```

Mehrdimensionale Arrays

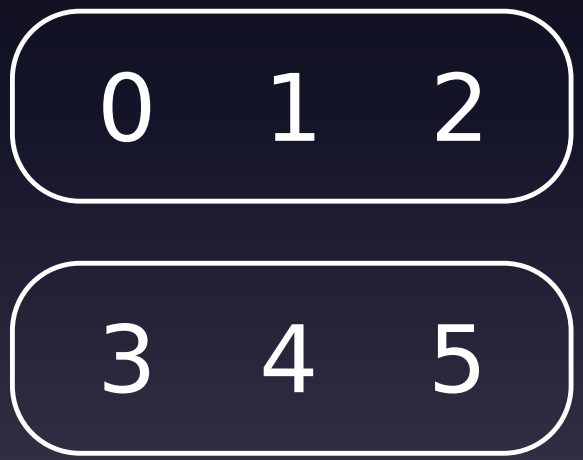
Mehrdimensionale Arrays

Zwei Varianten:

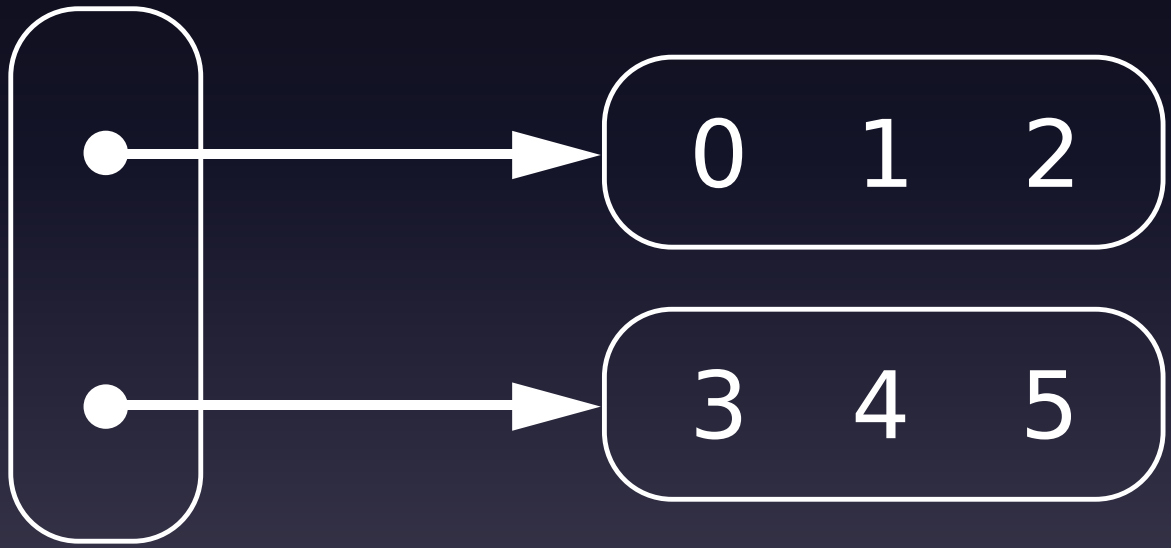
- Array von Arrays („deep array“)
- linear („flat array“)

Variante
„Array von Arrays“

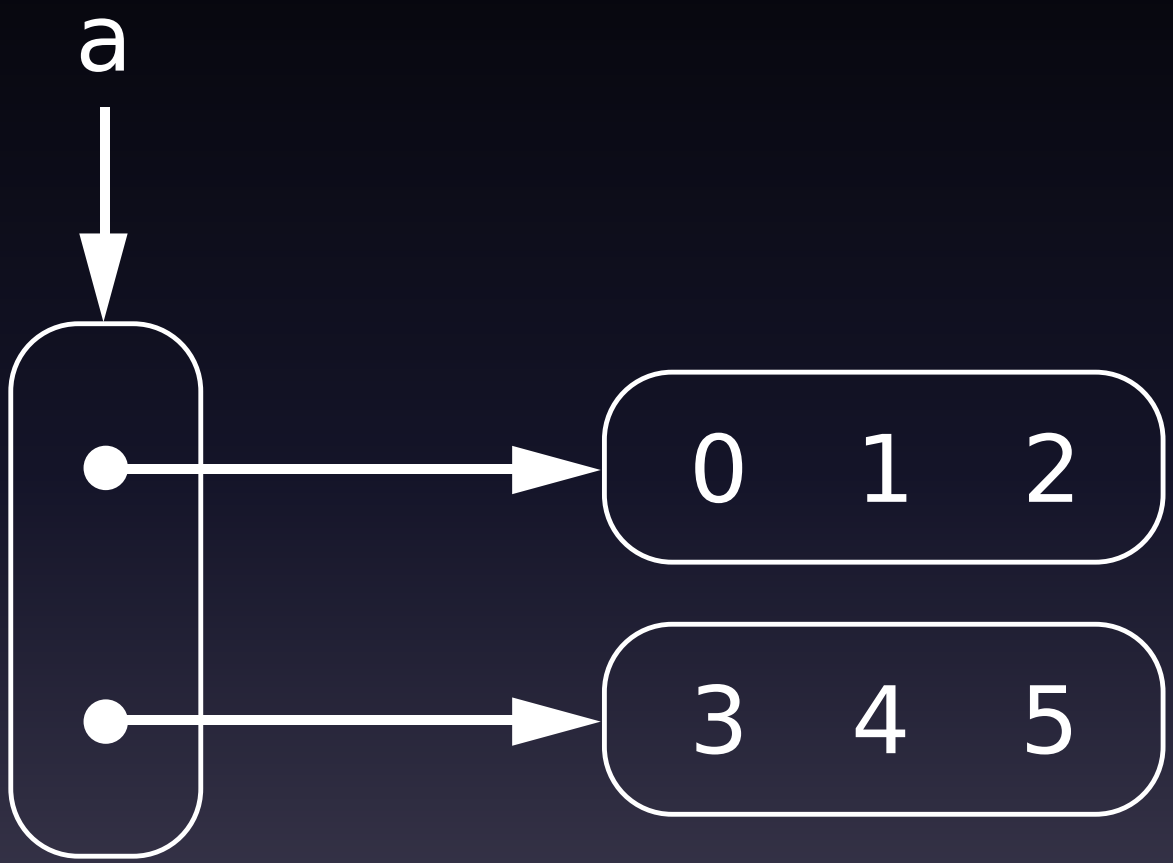
2D-Array als Array von Arrays



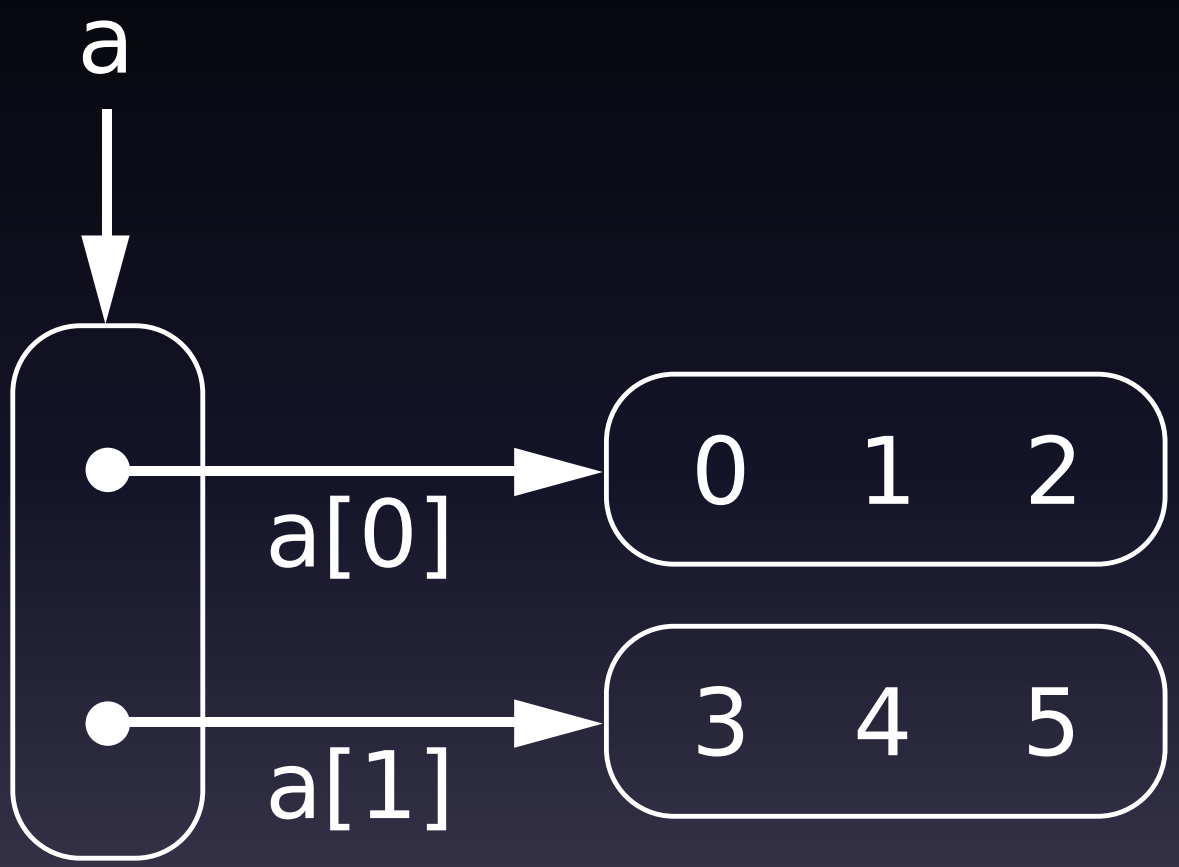
2D-Array als Array von Arrays



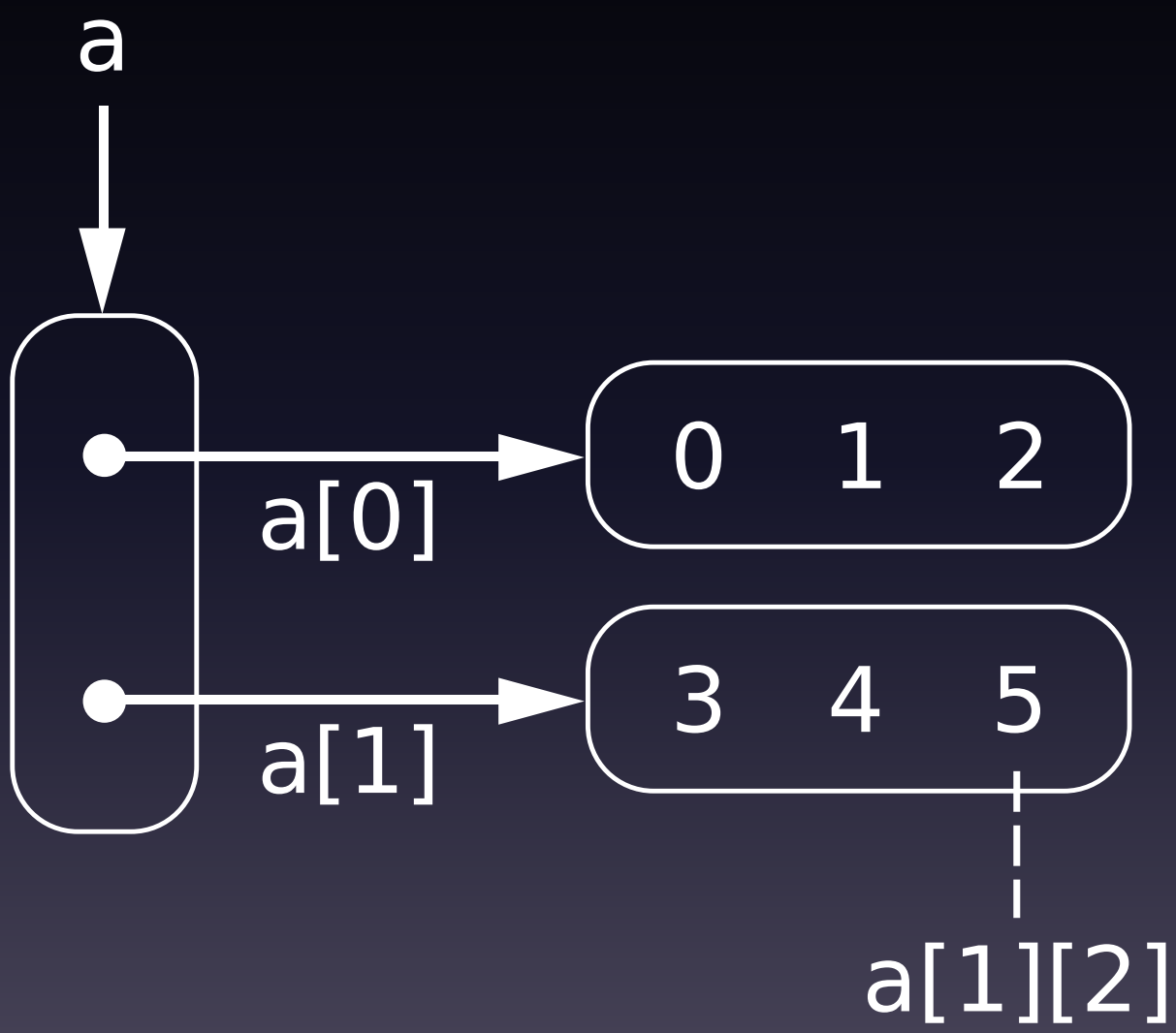
2D-Array als Array von Arrays



2D-Array als Array von Arrays



2D-Array als Array von Arrays



2D-Array als Array von Arrays

```
int main() {




  return 0;
}
```

2D-Array als Array von Arrays

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

  return 0;
}
```

2D-Array als Array von Arrays

```
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 Arrays

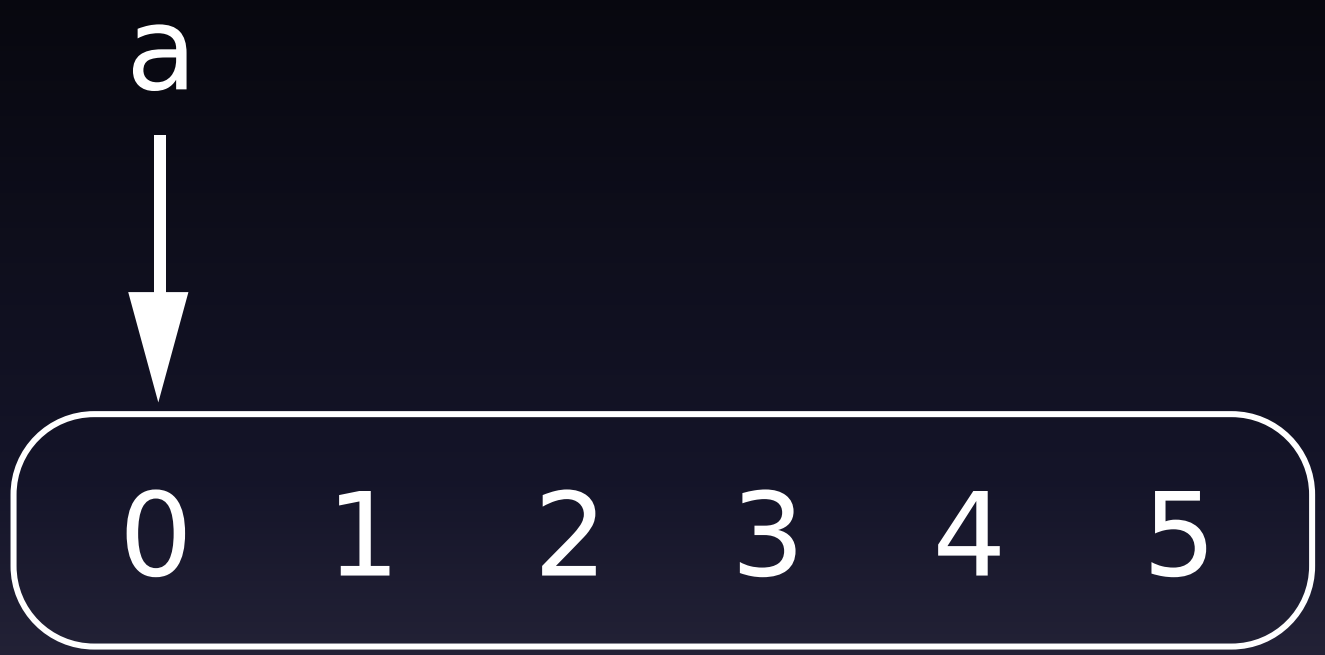
```
void demo_deep(int * const * a) {
  a[1][2] *= 2;
}

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

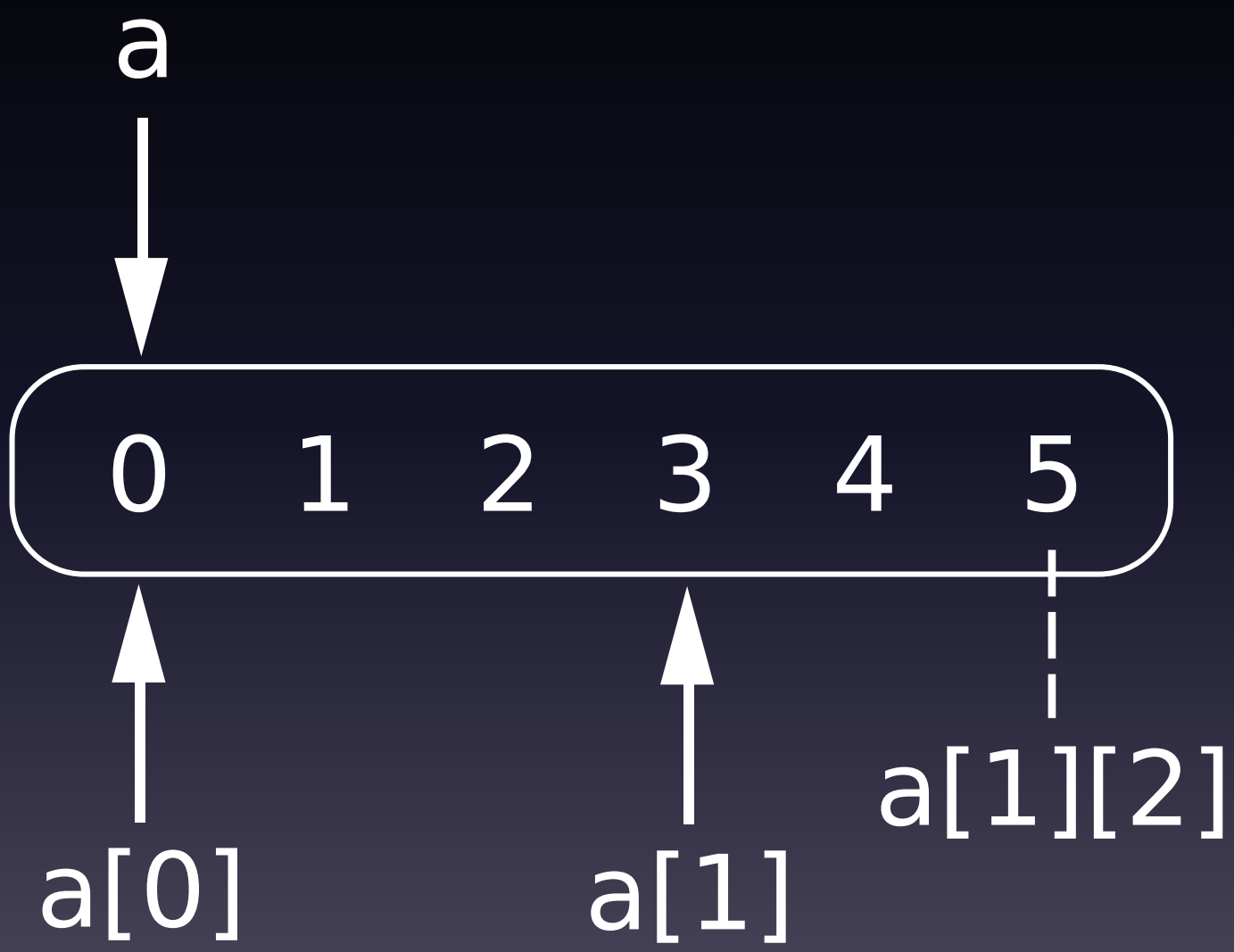
Variante „Linear“

2D-Array linear

2D-Array linear



2D-Array linear



2D-Array linear

```
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_deep(d);
  demo_flat(f);
  return 0;
}
```

2D-Array-Hybrid

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

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

`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
```

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 $*$ folgt einer Adresse
  (er *dereferenziert*)
- Pointer sind typisiert
- Arrays sind Pointer auf ihr erstes Element
- Strings sind nullterminiiniert
- Const Correctness bei Pointern ist wichtig

Danke!

Fragen?

http://blog.hartwork.org/