System-Level Programming

13 Pointers & Arrays

J. Kleinöder, D. Lohmann, V. Sieh, P. Wägemann

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg

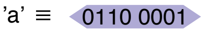
Summer Term 2025

http://sys.cs.fau.de/lehre/ss25

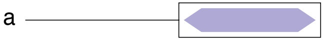
13-Zeiger_en

Classification: Pointers

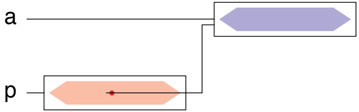
- **Literal:** `'a'`
  representation of a value



- **Variable:** `char a;`
  container for a value



- **Pointer variable:**
  `char *p = &a;`
  container for a reference
  to a variable



Pointers

- A *pointer* variable contains a memory address of a different variable as its value
  - A pointer points to another variable (in memory)
  - With the address, an indirect access to the target variable (its memory) is possible
- Therefore pointers are of **major relevance** for C programming
  - Functions now can change variables of the caller (*call by reference*) ↪ 9–5
  - Memory can be addressed directly
  - More efficient programs

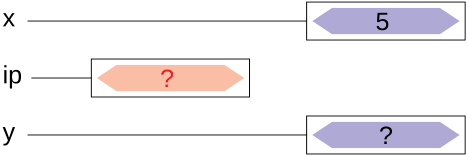
“Efficiency by machine orientation” ↪ 3–16

- However, pointers lead to many problems!
  - Structure of programs gets complicated (which functions can access which variables?)
  - Pointers are the **most common cause for errors** in C programs!

Definition of Pointer Variables

- **Pointer variable** := container for reference (↦ address)
- Syntax (definition): *type* ******identifier***;**
- Example

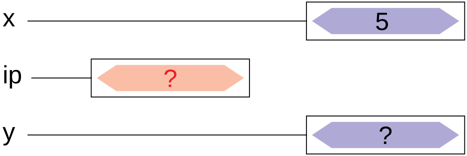
```
int x = 5;

int *ip;

int y;
```



Definition of Pointer Variables

- **Pointer variable** := container for reference (↦ address)
- Syntax (definition): *type* *****_identifier_**;**
- Example

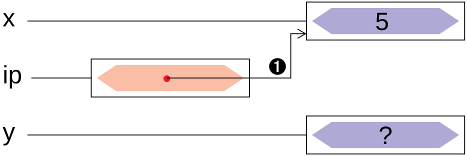
```
int x = 5;

int *ip;

int y;

ip = &x; ❶
```



Definition of Pointer Variables

- **Pointer variable** := container for reference (↦ address)
- Syntax (definition): *type* ***identifier*;**
- Example

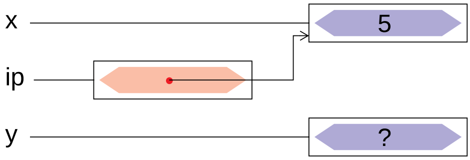
```
int x = 5;

int *ip;

int y;

ip = &x; ❶
```



Definition of Pointer Variables

- **Pointer variable** := container for reference (↦ address)
- Syntax (definition): *type* ******identifier*;
- Example

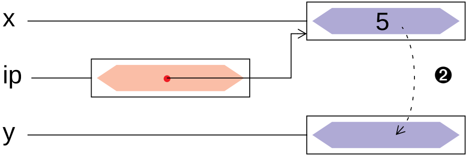
```
int x = 5;

int *ip;

int y;

ip = &x; ❶

y = *ip; ❷
```



Definition of Pointer Variables

- **Pointer variable** := container for reference (↦ address)
- Syntax (definition): *type* ******identifier***;**
- Example

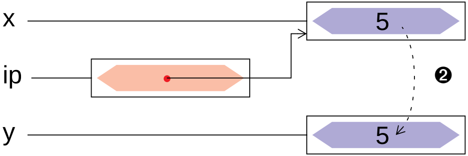
```
int x = 5;

int *ip;

int y;

ip = &x; ➊

y = *ip; ➋
```



Definition of Pointer Variables

- **Pointer variable** := container for reference (↦ address)
- Syntax (definition): *type* ******identifier***;**
- Example

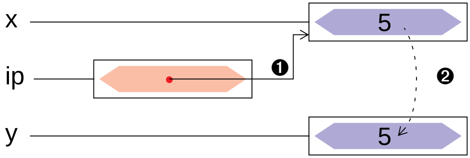
```
int x = 5;

int *ip;

int y;

ip = &x; ❶

y = *ip; ❷
```



Definition of Pointer Variables

- **Pointer variable** := container for reference (↦ address)
- Syntax (definition): *type* *****_identifier_**;**
- Example

```
int x = 5;

int *ip;

int y;

ip = &x; ❶

y = *ip; ❷
```



Address and Reference Operators

- *Address-of operator*: **&x** — The unary **&** operator provides the reference (↦ address in memory) of the variable **x**.

  also named *address operator*, memory aid: **&**ddress operator

- *Dereference operator*: **∗y** — The unary ∗ operator provides the target variable (↦ memory cell / container), to which the pointer **y** points (dereferencing).

- Valid code: **(∗(&x))** ≡ **x** — The reference operator is the inverse operation to the address operator.

Address and Reference Operators

- *Address-of operator*: **&x** — The unary **&** operator provides the reference (↦ address in memory) of the variable **x**.

  also named *address operator*, memory aid: **&**ddress operator

- *Dereference operator*: ***y** — The unary ∗ operator provides the target variable (↦ memory cell / container), to which the pointer **y** points (dereferencing).

- Valid code: **(∗(&x))** ≡ **x** — The reference operator is the inverse operation to the address operator.

Attention: Risk of Confusion (*** *I see stars everywhere* ***)

The ∗ symbol has different meanings in C **depending on the context**:

1. Multiplication (binary):	`x * y`	in expressions

2. Type modifier:	`uint8_t *p1, *p2`
	`typedef char *CPTR`
		in definitions and declarations

3. Reference (unary):	`x = *p1`	in expressions

In particular 2. and 3. often cause confusion

⤳ * is erroneously considered as part of the identifier.

Pointers as Function Arguments

- In C, parameters are always passed *by value* ↪ 9–5
  - Values of parameters are copied to local variables of the called function
  - The called function cannot change the actual parameters of the calling function

- This is also true for pointers (references)
  - The called function receives a copy of the address reference
  - With help of the ***** operators, the target variable can be accessed and its value can be changed

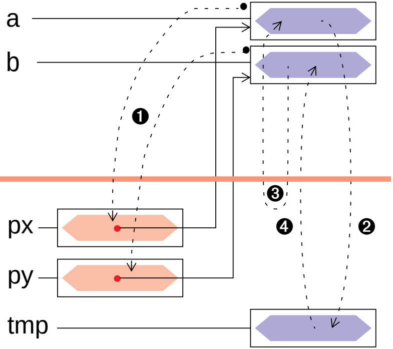
    ⇝ **call by reference**

Pointers as Function Arguments (continued)

- Example (overview)

```
void swap (int *, int *);
int main() {
    int a=47, b=11;
    ...
    swap(&a, &b); ❶
    ...
}
```

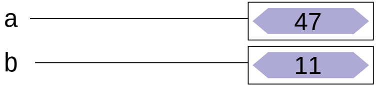
```
void swap (int *px, int *py)
{
    int tmp;

    tmp = *px; ❷
    *px = *py; ❸
    *py = tmp; ❹

}
```



Pointers as Function Arguments (continued)

- Example (step by step)

```
int main() {
    int a=47, b=11;
```

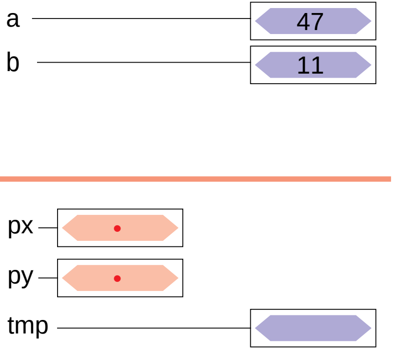


Pointers as Function Arguments (continued)

- Example (step by step)

```
void swap (int *, int *);
int main() {
    int a=47, b=11;
    ...
```

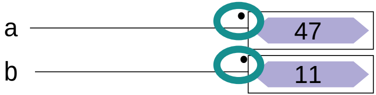
```
void swap (int *px, int *py)
{
    int tmp;
```



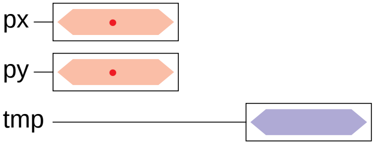
Pointers as Function Arguments (continued)

- Example (step by step)

```
    void swap (int *, int *);
    int main() {
        int a=47, b=11;
        ...
        swap(&a, &b);
```



```
    void swap (int *px, int *py)
    {
        int tmp;
```

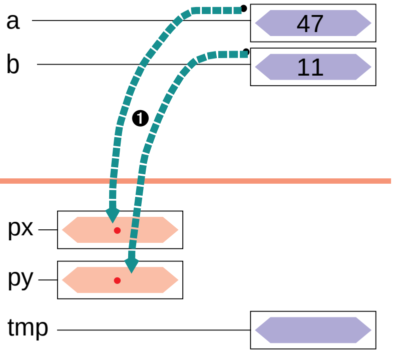


Pointers as Function Arguments (continued)

- Example (step by step)

```
void swap (int *, int *);
int main() {
    int a=47, b=11;
    ...
    swap(&a, &b); ❶
```

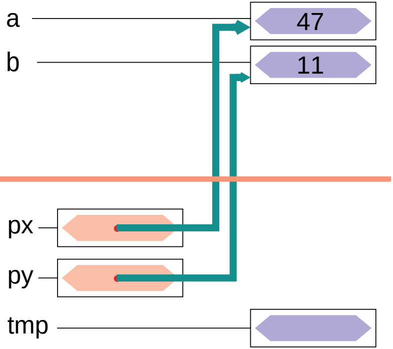
```
void swap (int *px, int *py)
{
    int tmp;
```



Pointers as Function Arguments (continued)

- Example (step by step)

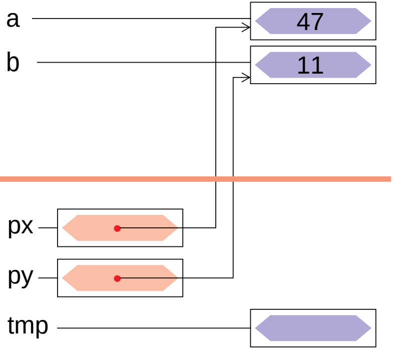
```
void swap (int *, int *);
int main() {
    int a=47, b=11;
    ...
    swap(&a, &b);


void swap (int *px, int *py)
{
    int tmp;
```



Pointers as Function Arguments (continued)

- Example (step by step)

```
    void swap (int *, int *);
    int main() {
        int a=47, b=11;
        ...
        swap(&a, &b);

    void swap (int *px, int *py)
    {
        int tmp;

        tmp = *px; ❷
```

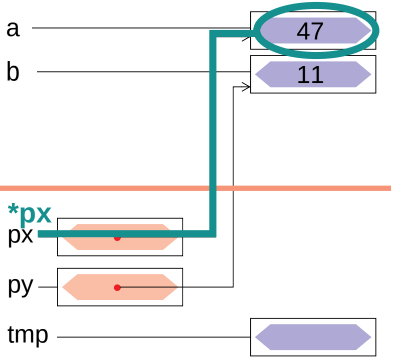


Pointers as Function Arguments (continued)

- Example (step by step)

```
    void swap (int *, int *);
    int main() {
        int a=47, b=11;
        ...
        swap(&a, &b);
```

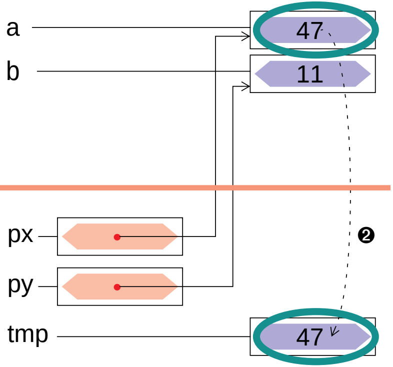
```
    void swap (int *px, int *py)
    {
        int tmp;

        tmp = *px; ❷
```



Pointers as Function Arguments (continued)

- Example (step by step)

```
void swap (int *, int *);
int main() {
    int a=47, b=11;
    ...
    swap(&a, &b);

void swap (int *px, int *py)
{
    int tmp;

    tmp = *px; ❷
```

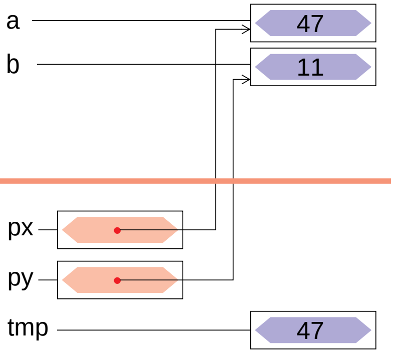


Pointers as Function Arguments (continued)

- Example (step by step)

```
    void swap (int *, int *);
    int main() {
        int a=47, b=11;
        ...
        swap(&a, &b);
```

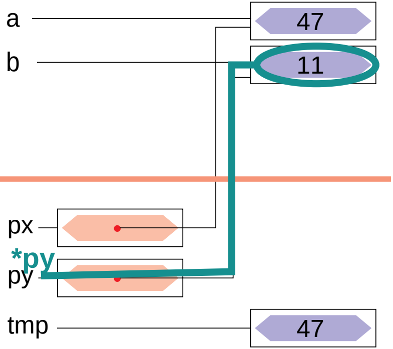
```
    void swap (int *px, int *py)
    {
        int tmp;

        tmp = *px; ❷
        *px = *py; ❸
```



Pointers as Function Arguments (continued)

- Example (step by step)

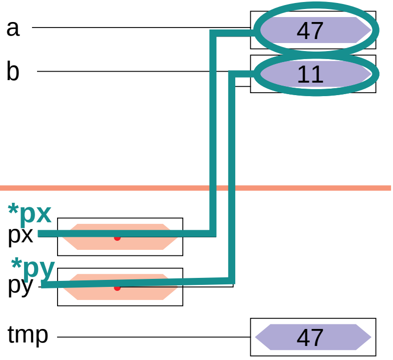
```
void swap (int *, int *);
int main() {
    int a=47, b=11;
    ...
    swap(&a, &b);

void swap (int *px, int *py)
{
    int tmp;

    tmp = *px; ❷
    *px = *py; ❸
```



Pointers as Function Arguments (continued)

- Example (step by step)

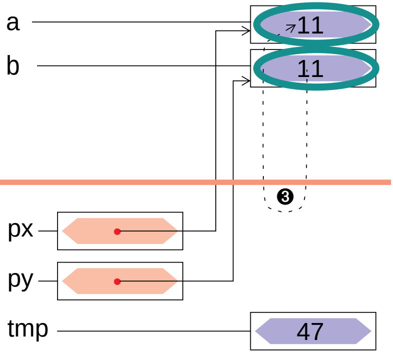
```
   void swap (int *, int *);
   int main() {
       int a=47, b=11;
       ...
       swap(&a, &b);


   void swap (int *px, int *py)
   {
       int tmp;

       tmp = *px; ❷
       *px = *py; ❸
```



Pointers as Function Arguments (continued)

- Example (step by step)

```
void swap (int *, int *);
int main() {
    int a=47, b=11;
    ...
    swap(&a, &b);

void swap (int *px, int *py)
{
    int tmp;

    tmp = *px; ❷
    *px = *py; ❸
```

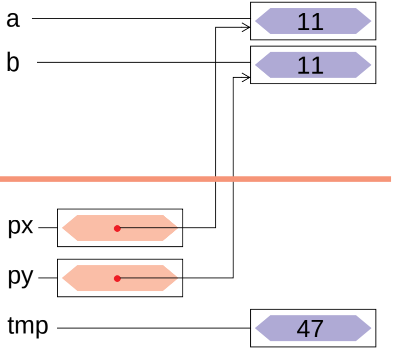


Pointers as Function Arguments (continued)

- Example (step by step)

```
void swap (int *, int *);
int main() {
    int a=47, b=11;
    ...
    swap(&a, &b);
```

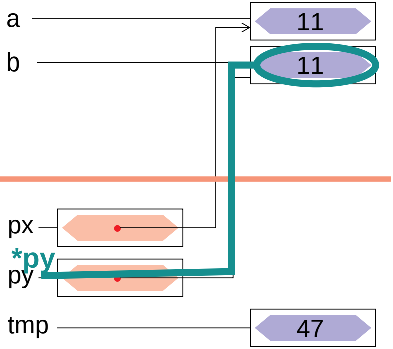
```
void swap (int *px, int *py)
{
    int tmp;

    tmp = *px; ❷
    *px = *py; ❸
    *py = tmp; ❹

}
```



Pointers as Function Arguments (continued)

- Example (step by step)

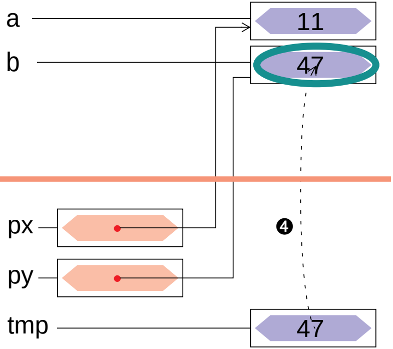
```
void swap (int *, int *);
int main() {
    int a=47, b=11;
    ...
    swap(&a, &b);

void swap (int *px, int *py)
{
    int tmp;

    tmp = *px; ❷
    *px = *py; ❸
    *py = tmp; ❹

}
```



Pointers as Function Arguments (continued)

- Example (step by step)

```
void swap (int *, int *);
int main() {
    int a=47, b=11;
    ...
    swap(&a, &b);


void swap (int *px, int *py)
{
    int tmp;

    tmp = *px; ❷
    *px = *py; ❸
    *py = tmp; ❹

}
```



Arrays [≈Python]

- **Array variable** := container for a list of values of same type
- Syntax (definition): *type identifier* **[** *IntExpression* **] ;**
  - *type* — type of the values [=Python]
  - *identifier* — name of the array variable [=Python]
  - *IntExpression* — **constant** integer expression, defines the size of the array (↦ number of elements). [≠Python]

    From **C99** onwards, the *IntExpression* of `auto` arrays can be chosen **variably** (i. e., arbitrary, but constant).
- Example:

```c
static uint8_t LEDs[8 * 2];     // constant, fixed array size

void f(int n) {
  auto char a[NUM_LEDS * 2];    // constant, fixed array size
  auto char b[n];               // C99: variable, fixed array size
}
```

Array Initialization

- Like other variables, an array can receive a set of initial values during definition

```
uint8_t LEDs[4] = { RED0, YELLOW0, GREEN0, BLUE0 };
int prim[5]     = { 1, 2, 3, 5, 7 };
```

Array Initialization

- Like other variables, an array can receive a set of initial values during definition

```
uint8_t LEDs[4] = { RED0, YELLOW0, GREEN0, BLUE0 };
int prim[5]     = { 1, 2, 3, 5, 7 };
```

- If not all initializing elements are given, the remainder is initialized with 0

```
uint8_t LEDs[4] = { RED0 };     // => { RED0, 0, 0, 0 }
int prim[5]     = { 1, 2, 3 };  // => { 1, 2, 3, 0, 0 }
```

Array Initialization

- Like other variables, an array can receive a set of initial values during definition

```
uint8_t LEDs[4] = { RED0, YELLOW0, GREEN0, BLUE0 };
int prim[5]     = { 1, 2, 3, 5, 7 };
```

- If not all initializing elements are given, the remainder is initialized with 0

```
uint8_t LEDs[4] = { RED0 };     // => { RED0, 0, 0, 0 }
int prim[5]     = { 1, 2, 3 };  // => { 1, 2, 3, 0, 0 }
```

- If the explicit dimension of the array is omitted, the number of initializing elements determines the size

```
uint8_t LEDs[]  = { RED0, YELLOW0, GREEN0, BLUE0 };
int prim[]      = { 1, 2, 3, 5, 7 };
```

Access to Arrays

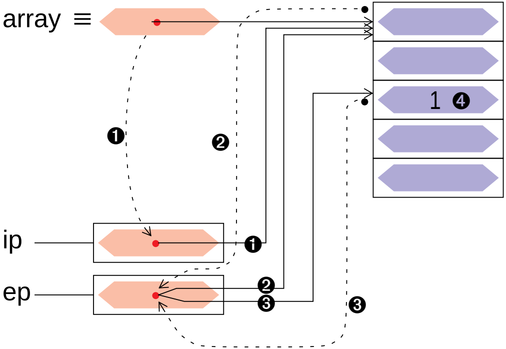
- Syntax: *array* **[** *IntExpression* **]** [=Python]
  - With $0 \leq \mathit{IntExpression} < n$ for $n =$ size of the array
  - **Attention:** The index is not checked [≠Python]
    ⇝ common cause for errors in C programs
- Example

```c
uint8_t LEDs[] = { RED0, YELLOW0, GREEN0, BLUE0 };

LEDs[3] = BLUE1;

for (uint8_t i = 0; i < 4; i++) {
  sb_led_on(LEDs[i]);
}

LEDs[4] = GREEN1;   // UNDEFINED!!!
```

Arrays are Pointers

- The identifier of an array is syntactically equivalent to a constant pointer to the first element of the array: `array` $\equiv$ `&array[0]`
  - An alias – not a container ⇝ value cannot be changed
  - Via such a pointer, the indirect access to array cells is possible
- Example (overview)

```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸

*ep = 1; ❹
```

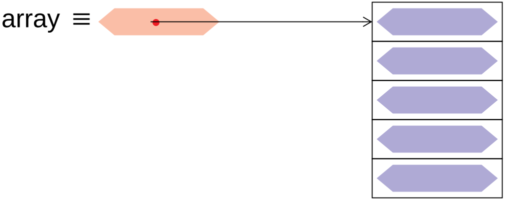


Arrays are Pointers

- The identifier of an array is syntactically equivalent to a constant pointer to the first element of the array: `array` ≡ `&array[0]`
  - An alias – not a container ⇝ value cannot be changed
  - Via such a pointer, the indirect access to array cells is possible

- Example (step by step)

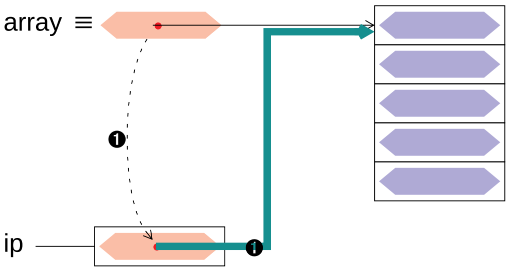
```
int array[5];
```



Arrays are Pointers

- The identifier of an array is syntactically equivalent to a constant pointer to the first element of the array: `array` ≡ `&array[0]`
  - An alias – not a container ⇝ value cannot be changed
  - Via such a pointer, the indirect access to array cells is possible
- Example (step by step)

```
int array[5];

int *ip = array; ❶
```

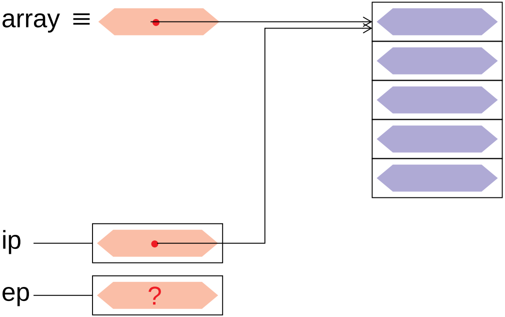


Arrays are Pointers

- The identifier of an array is syntactically equivalent to a constant pointer to the first element of the array: `array` ≡ `&array[0]`
  - An alias – not a container ⇝ value cannot be changed
  - Via such a pointer, the indirect access to array cells is possible

- Example (step by step)

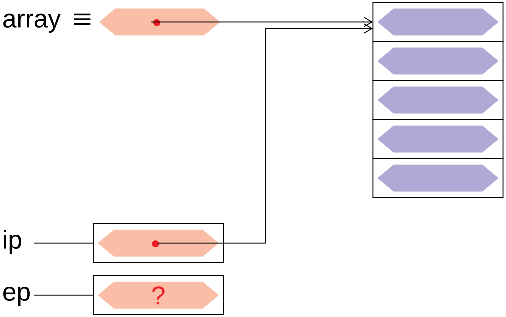
```
int array[5];

int *ip = array; ➊

int *ep;
```



Arrays are Pointers

- The identifier of an array is syntactically equivalent to a constant pointer to the first element of the array: `array` ≡ `&array[0]`
  - An alias – not a container ⇝ value cannot be changed
  - Via such a pointer, the indirect access to array cells is possible
- Example (step by step)

```
int array[5];

int *ip = array; ➊

int *ep;
ep = &array[0]; ➋
```

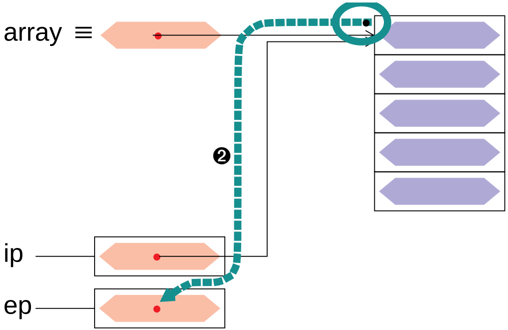


Arrays are Pointers

- The identifier of an array is syntactically equivalent to a constant pointer to the first element of the array: `array` ≡ `&array[0]`
  - An alias – not a container ⇝ value cannot be changed
  - Via such a pointer, the indirect access to array cells is possible

- Example (step by step)

```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷
```

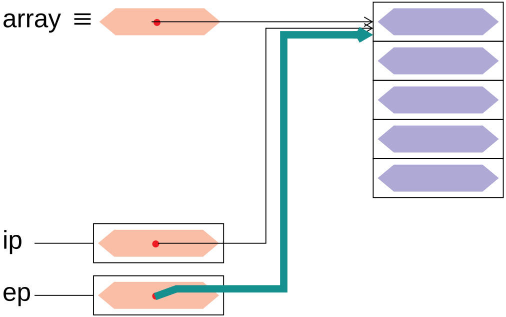


Arrays are Pointers

- The identifier of an array is syntactically equivalent to a constant pointer to the first element of the array: `array` ≡ `&array[0]`
  - An alias – not a container ⇝ value cannot be changed
  - Via such a pointer, the indirect access to array cells is possible

- Example (step by step)

```
int array[5];

int *ip = array; ➊

int *ep;
ep = &array[0]; ❷
```

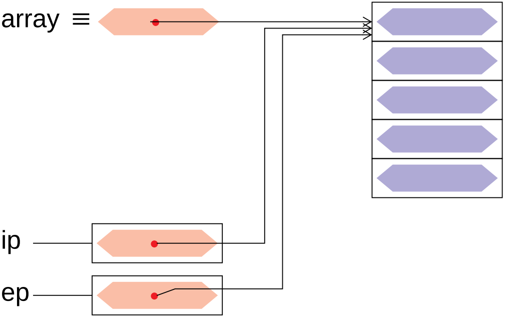


Arrays are Pointers

- The identifier of an array is syntactically equivalent to a constant pointer to the first element of the array: `array` ≡ `&array[0]`
  - An alias – not a container ⤳ value cannot be changed
  - Via such a pointer, the indirect access to array cells is possible

- Example (step by step)

```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸
```

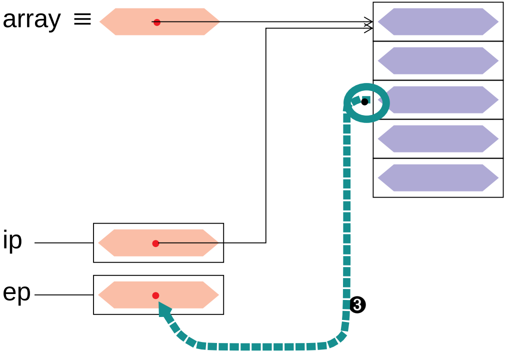


Arrays are Pointers

- The identifier of an array is syntactically equivalent to a constant pointer to the first element of the array: `array` ≡ `&array[0]`
  - An alias – not a container ⇝ value cannot be changed
  - Via such a pointer, the indirect access to array cells is possible

- Example (step by step)

```
int array[5];

int *ip = array; ➊

int *ep;
ep = &array[0]; ➋

ep = &array[2]; ❸
```

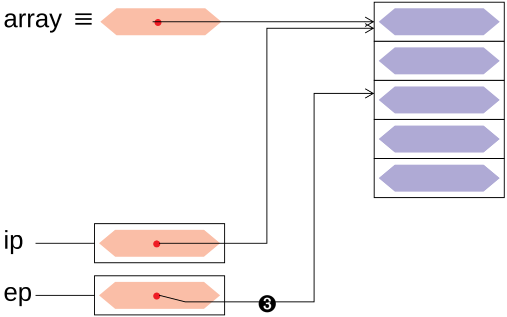


Arrays are Pointers

- The identifier of an array is syntactically equivalent to a constant pointer to the first element of the array: `array` ≡ `&array[0]`
  - An alias – not a container ⇝ value cannot be changed
  - Via such a pointer, the indirect access to array cells is possible

- Example (step by step)

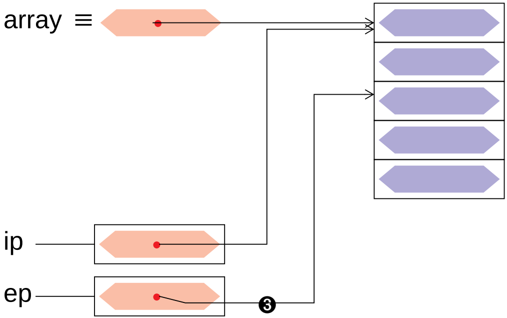
```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸
```



Arrays are Pointers

- The identifier of an array is syntactically equivalent to a constant pointer to the first element of the array: `array` ≡ `&array[0]`
  - An alias – not a container ⇝ value cannot be changed
  - Via such a pointer, the indirect access to array cells is possible
- Example (step by step)

```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸

*ep = 1; ❹
```

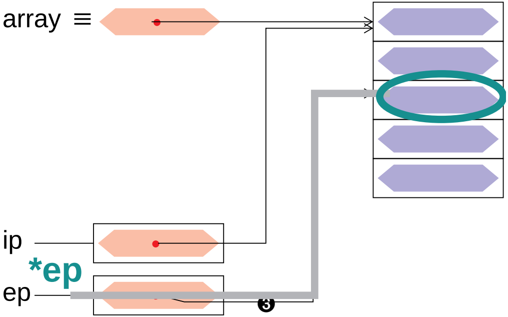


Arrays are Pointers

- The identifier of an array is syntactically equivalent to a constant pointer to the first element of the array: `array` ≡ `&array[0]`
  - An alias – not a container ⇝ value cannot be changed
  - Via such a pointer, the indirect access to array cells is possible

- Example (step by step)

```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸

*ep = 1; ❹
```

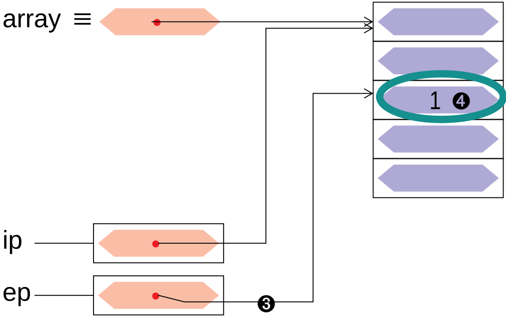


Arrays are Pointers

- The identifier of an array is syntactically equivalent to a constant pointer to the first element of the array: `array` ≡ `&array[0]`
  - An alias – not a container ⤳ value cannot be changed
  - Via such a pointer, the indirect access to array cells is possible

- Example (step by step)

```
int array[5];

int *ip = array; ➊

int *ep;
ep = &array[0]; ➋

ep = &array[2]; ➌

*ep = 1; ➍
```



Pointers are Arrays

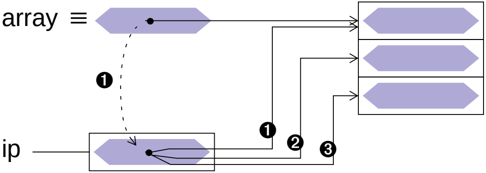
- The identifier of an array is syntactically equivalent to a constant pointer to the first element of the array: `array` $\equiv$ `&array[0]`
- This relation is valid in both directions: `*array` $\equiv$ `array[0]`
  - A pointer can be used like an array
  - In particular, the **[ ]**-operator can be used ↪ 13–9
- Example (see ↪ 13–9)

```
uint8_t LEDs[] = { RED0, YELLOW0, GREEN0, BLUE0 };

LEDs[3]    = BLUE1;
uint8_t *p = LEDs;

for (unit8_t i = 0; i < 4; i++) {
  sb_led_on(p[i]);
}
```

Arithmetic Operations on Pointers

- In contrast to the identifier of an array, a pointer *variable* is a container ⇝ its value can be modified
- Besides simple assignments, arithmetic operations are possible

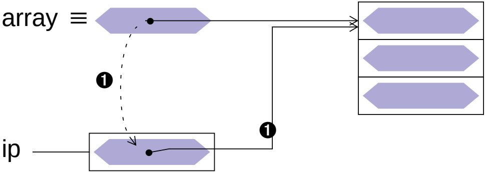
```
int array[3];
int *ip = array; ❶

ip++; ❷
ip++; ❸
```



Arithmetic Operations on Pointers

- In contrast to the identifier of an array, a pointer *variable* is a container ⇝ its value can be modified
- Besides simple assignments, arithmetic operations are possible

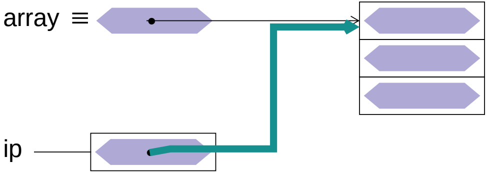
```
int array[3];
int *ip = array; ❶
```



Arithmetic Operations on Pointers

- In contrast to the identifier of an array, a pointer *variable* is a container ⤳ its value can be modified
- Besides simple assignments, arithmetic operations are possible

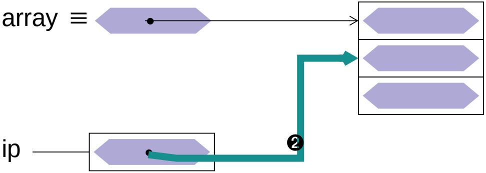
```
int array[3];
int *ip = array; ❶

ip++; ❷
```



Arithmetic Operations on Pointers

- In contrast to the identifier of an array, a pointer *variable* is a container ⤳ its value can be modified
- Besides simple assignments, arithmetic operations are possible

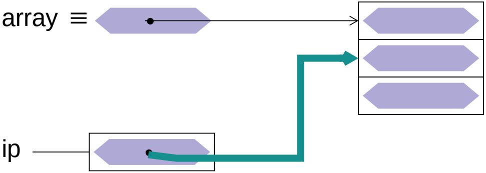
```
int array[3];
int *ip = array; ➊

ip++; ➋
```



Arithmetic Operations on Pointers

- In contrast to the identifier of an array, a pointer *variable* is a container ⤳ its value can be modified
- Besides simple assignments, arithmetic operations are possible

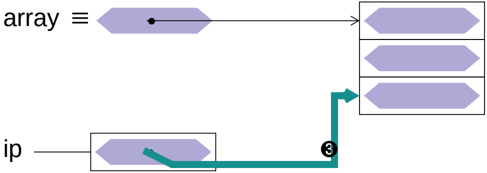
```
int array[3];
int *ip = array; ❶

ip++; ❷
ip++; ❸
```



Arithmetic Operations on Pointers

- In contrast to the identifier of an array, a pointer *variable* is a container ⇝ its value can be modified
- Besides simple assignments, arithmetic operations are possible

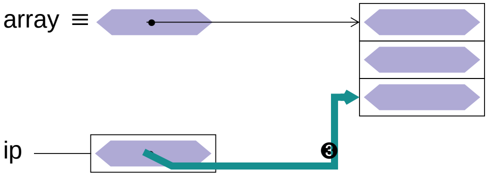
```
int array[3];
int *ip = array; ❶

ip++; ❷
ip++; ❸
```



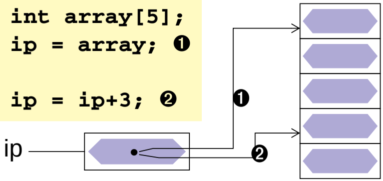
Arithmetic Operations on Pointers

- In contrast to the identifier of an array, a pointer *variable* is a container ⇝ its value can be modified
- Besides simple assignments, arithmetic operations are possible

```
int array[3];
int *ip = array; ❶

ip++; ❷
ip++; ❸
```



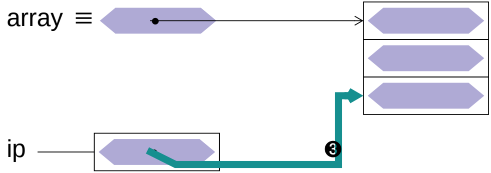
```
int array[5];
ip = array; ❶

ip = ip+3; ❷
```



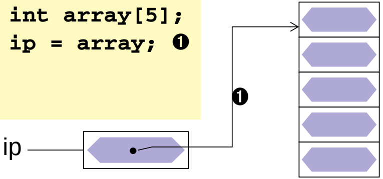
Arithmetic Operations on Pointers

- In contrast to the identifier of an array, a pointer *variable* is a container ⤳ its value can be modified
- Besides simple assignments, arithmetic operations are possible

```
int array[3];
int *ip = array; ❶

ip++; ❷
ip++; ❸
```



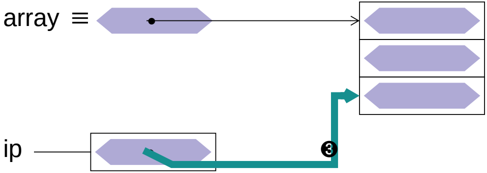
```
int array[5];
ip = array; ❶
```



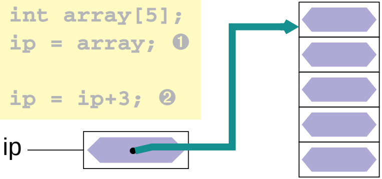
Arithmetic Operations on Pointers

- In contrast to the identifier of an array, a pointer *variable* is a container ⤳ its value can be modified
- Besides simple assignments, arithmetic operations are possible

```
int array[3];
int *ip = array; ❶

ip++; ❷
ip++; ❸
```



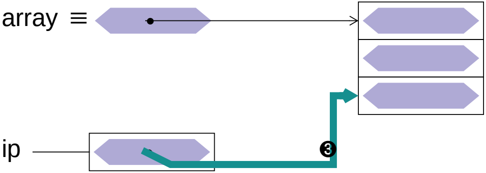
```
int array[5];
ip = array; ❶

ip = ip+3; ❷
```



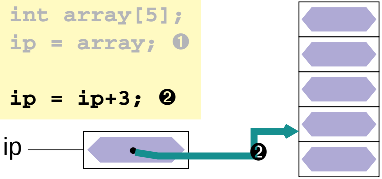
Arithmetic Operations on Pointers

- In contrast to the identifier of an array, a pointer *variable* is a container ⇝ its value can be modified
- Besides simple assignments, arithmetic operations are possible

```
int array[3];
int *ip = array; ❶

ip++; ❷
ip++; ❸
```



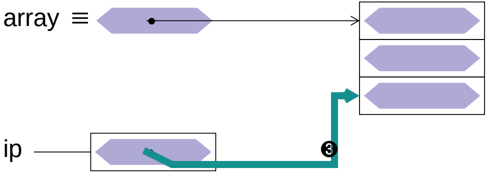
```
int array[5];
ip = array; ❶

ip = ip+3; ❷
```



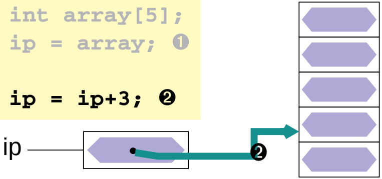
Arithmetic Operations on Pointers

- In contrast to the identifier of an array, a pointer *variable* is a container ⇝ its value can be modified
- Besides simple assignments, arithmetic operations are possible

```
int array[3];
int *ip = array; ❶

ip++; ❷
ip++; ❸
```



```
int array[5];
ip = array; ❶

ip = ip+3; ❷
```



(ip+3) ≡ &ip[3]

When using arithmetic operations on pointers, the size of the type of one object is always taken into account.

Pointer Arithmetic – Operations

- Arithmetic operations

  ++ pre/post increment
  ⇝ shift to the next object

  −− pre/post decrement
  ⇝ shift to previous object

  +, − addition / subtraction of an `int` value
  ⇝ resulting pointer is moved by *n* objects

  − subtraction of two pointers
  ⇝ number of objects *n* between the pointers (distance)

- Comparison operators: $<$, $<=$, $==$, $>=$, $>$, $!=$

  ↪ 7–3

  ⇝ pointers can be compared and ordered like integers

Arrays are Pointers are Arrays – Summary

- In combination with arithmetic operations for pointers, each array operation can be mapped to an equivalent pointer operation.

- For `int i, array[N], *ip = array;` with $0 \le i < N$ holds:

`array`	≡	`&array[0]`	≡	`ip`	≡	`&ip[0]`
`*array`	≡	`array[0]`	≡	`*ip`	≡	`ip[0]`
`*(array + i)`	≡	`array[i]`	≡	`*(ip + i)`	≡	`ip[i]`
		`array++`	≢	`ip++`		

Error: `array` is constant!

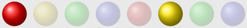
- In contrary, pointer operations can be represented by array operations.

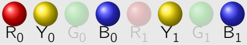
  However, the identifier of the array cannot be modified.

Arrays as Function Arguments

- Arrays are **always** passed as pointers in C

```
static uint8_t LEDs[] = { RED0, YELLOW1 };

void enlight(uint8_t *array, unsigned n) {
  for (unsigned i = 0; i < n; i++)
    sb_led_on(array[i]);
}

void main() {
  enlight(LEDs, 2);
  uint8_t moreLEDs[] = { YELLOW0, BLUE0, BLUE1 };
  enlight(moreLEDs, 3);
}
```





- Information on size of the array is lost!
  - The size has to passed explicitly as another parameter
  - In some cases, the size can be calculated inside the function (e. g., by searching for the terminating `NUL` symbol at the end of a string)

Arrays as Function Arguments (continued)

- Arrays are **always** passed as pointers in C
- If the parameter is declared as `const`, the function cannot modify the elements of the array ↦ good style!

```c
void enlight(const uint8_t *array, unsigned n) {
  ...
}
```

Arrays as Function Arguments (continued)

- Arrays are **always** passed as pointers in C
- If the parameter is declared as `const`, the function cannot modify the elements of the array ↦ good style!

```
void enlight(const uint8_t *array, unsigned n) {
  ...
}
```

- To clarify, that an array (and not a “pointer to a variable”) is expected, one can use the following equivalent syntax:

```
void enlight(const uint8_t array[], unsigned n) {
  ...
}
```

Arrays as Function Arguments (continued)

- Arrays are **always** passed as pointers in C
- If the parameter is declared as `const`, the function cannot modify the elements of the array ↦ good style!

```c
void enlight(const uint8_t *array, unsigned n) {
  ...
}
```

- To clarify, that an array (and not a “pointer to a variable”) is expected, one can use the following equivalent syntax:

```c
void enlight(const uint8_t array[], unsigned n) {
  ...
}
```

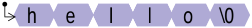
- Attention: This is only valid for declaring function parameters
- For defining variables, `array[]` has a entirely different meaning (identifying size of the array from list of initializers ↪ 13–8)

Arrays as Function Arguments (continued)

- The function **int strlen(const char *)** from the standard library provides the number of characters of the passed string

```
void main() {
  ...
  const char *string = "hello";   // string is array of char
  sb_7seg_showNumber(strlen(string));
  ...
}
```

It holds: "hello" ≡



h e l l o \0

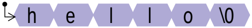
↪ 6–13

Arrays as Function Arguments (continued)

- The function `int strlen(const char *)` from the standard library provides the number of characters of the passed string

```
void main() {
  ...
  const char *string = "hello";   // string is array of char
  sb_7seg_showNumber(strlen(string));
  ...
}
```

It holds: "hello" ≡



↪ 6–13

- Variants of implementation

option 1: array syntax

```
int strlen(const char s[]) {
  int n = 0;
  while (s[n] != '\0')
    n++;
  return n;
}
```

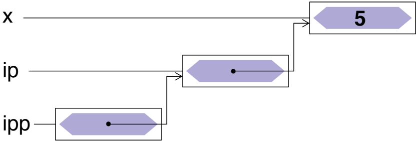
option 2: pointer syntax

```
int strlen(const char *s) {
  const char *end = s;
  while (*end != '\0')
    end++;
  return end - s;
}
```

Pointers to Pointers

- A pointer can point to another pointer variable

```
int x = 5;
int *ip = &x;

int **ipp = &ip;
/* → **ipp = 5 */
```



- This is particularly useful for passing parameters to functions
  - pointer parameter is passed *call by reference* (e. g., `swap()` function for pointers)
  - passing an array of pointers

Pointers to Functions

- A pointer can point to a function
  - With this feature, functions are passed as parameters to other functions
    ↦ functions of higher order
- Example

```c
// invokes job() every second
void doPeriodically(void (*job)(void)) {
  while (1) {
    job();      // invoke job
    for (volatile uint16_t i = 0; i < 0xffff; i++)
    ;         // wait a second
  }
}

void blink(void) {
  sb_led_toggle(RED0);
}

void main() {
  doPeriodically(blink);  // pass blink() as parameter
}
```

Pointers to Functions (continued)

- Syntax (definition): *type* **(* *identifier*)(** *formalParam*$_{opt}$ **);**
  (similar to function definitions) ↪ 9–3
  - *type* return value of the functions the pointer can point to
  - *identifier* name of the function pointer
  - *formalParam*$_{opt}$ formal parameters of the functions the pointer can point to: *type*$_1$,..., *type*$_n$
- A function pointer is used in the same way as a function
  - call with *identifier*(*actParam*) ↪ 9–4
  - address (&) and reference operator (*) are not required ↪ 13–4
  - an identifier of a function is a constant pointer to that function

```
void blink(uint8_t which) { sb_led_toggle(which); }

void main() {
  void (*myfun)(uint8_t); // myfun is pointer to function
  myfun = blink;          // blink is constant pointer to function
  myfun(RED0);            // invoke blink() via function pointer
  blink(RED0);            // invoke blink()
}
```

13-Zeiger_en

Pointers to Functions (continued)

- Function pointers are often used for callback functions to deliver asynchronous events (↦ “listener” pattern)

```c
// Example: asynchronous button events with libspicboard
#include <avr/interrupt.h>         // for sei()
#include <7seg.h>                  // for sb_7seg_showNumber()
#include <button.h>                // for button stuff

// callback handler for button events (invoked on interrupt level)
void onButton(BUTTON b, BUTTONEVENT e) {
  static int8_t count = 1;
  sb_7seg_showNumber(count++);    // show no of button presses
  if (count > 99) count = 1;      // reset at 100
}

void main() {
  sb_button_registerCallback(     // register callback
    BUTTON0, BUTTONEVENT_PRESSED, //   for this button and events
    onButton                      //   invoke this function
    );
  sei();                          // enable interrupts (necessary!)
  while (1) {}                    // wait forever
}
```

Summary

- A pointer references a variable in memory
  - possibility for indirect access to a value
  - basis for implementation of *call-by-reference* in C
  - basis for implementation of arrays
  - **important part of the machine orientation** of C
  - **most common cause for errors in C programs!**
- The syntactical possibilities are diverse (and confusing)
  - type modifier `*`, address operator `&`, reference operator `*`
  - pointer arithmetic with `+`, `-`, `++`, and `--`
  - syntactical equivalence between pointers and arrays (`[]` Operator)
- Pointers can point to functions
  - pass functions to functions
  - principle of callback functions