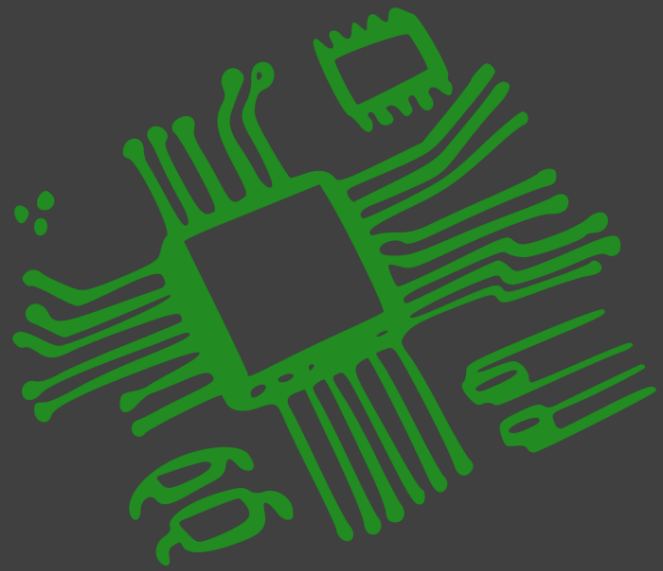
Computer Engineering, VU

(448.012)

— Unit 6: Arrays and Strings —

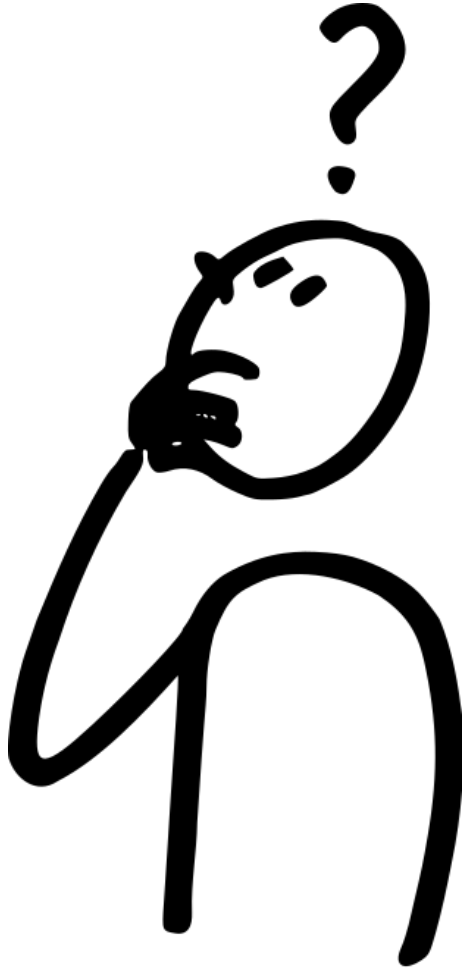


Ass.Prof. Dr. Tobias Scheipel
tobias.scheipel@tugraz.at

Reconfigurable Computer Architectures
Institute of Technical Informatics
Graz University of Technology

General Overview

- What is an array? Why do I need arrays?
  - How to define, declare, and initialize arrays
  - How to access array elements
  - Multi-dimensional arrays
- What is a string? Why do I need strings?
  - How to define, declare, and initialize strings
  - Common string functions
  - Arrays of strings



What is an array? Why do I need arrays?

main.c

```c
void main() {
  float temp_august_1, temp_august_2, temp_august_3, temp_august_4, temp_august_5, temp_august_6,
       temp_august_7, temp_august_8, temp_august_9, temp_august_10, temp_august_11, temp_august_12,
       temp_august_13, temp_august_14, temp_august_15, temp_august_16, temp_august_17, temp_august_18,
       temp_august_19, temp_august_20, temp_august_21, temp_august_22, temp_august_23, temp_august_24,
       temp_august_25, temp_august_26, temp_august_27, temp_august_28, temp_august_29, temp_august_30,
       temp_august_31;

  uint16 current_day;

  [...] // code to set current_day and read temperature values

  switch (current_day) {
    case 1: // process temp_august_1
      break;
    case 2: // process temp_august_2
      break;     [...]
    default:
      // handle invalid day
      break;
  } [...] // further code
}
```

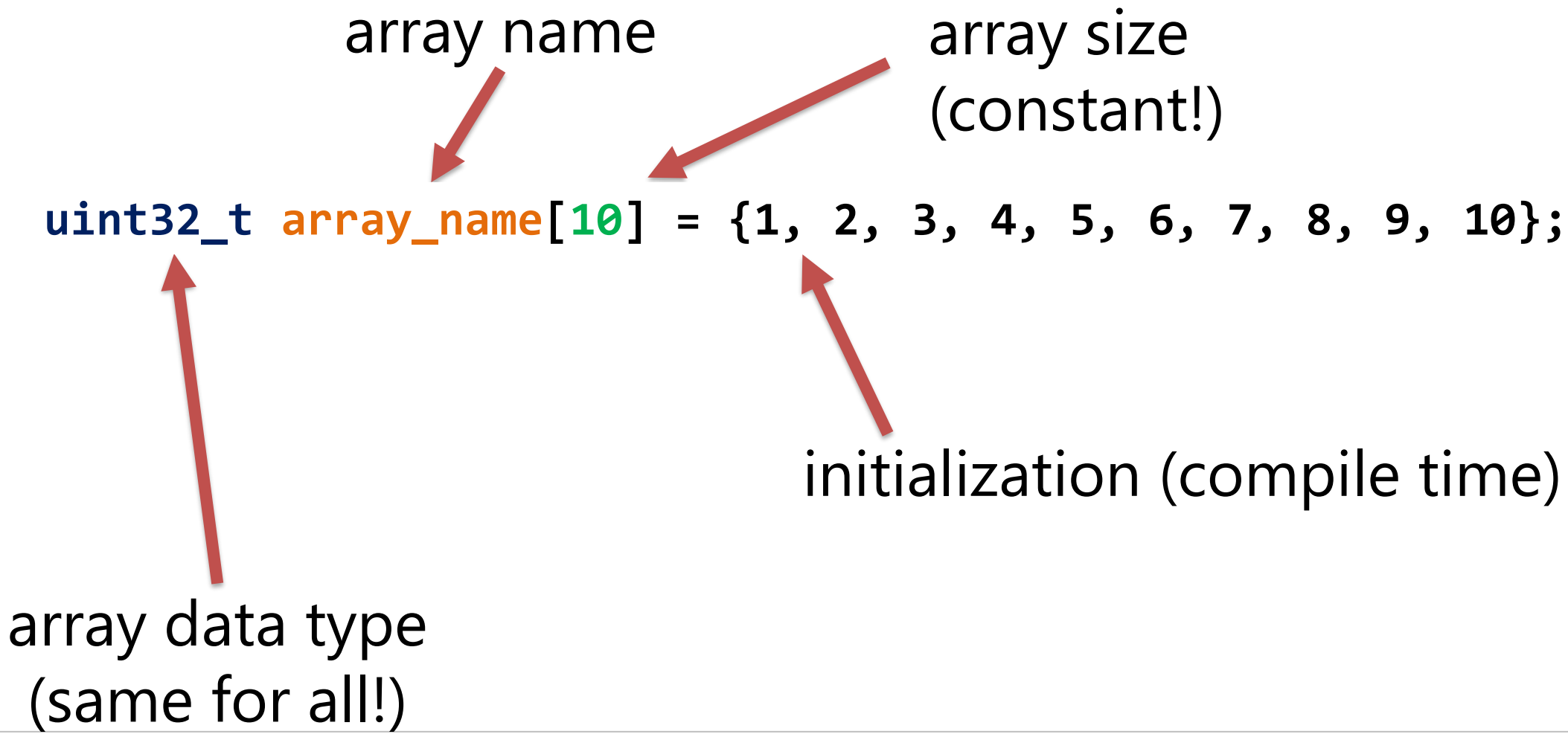
What is an array? Why do I need arrays?

main.c

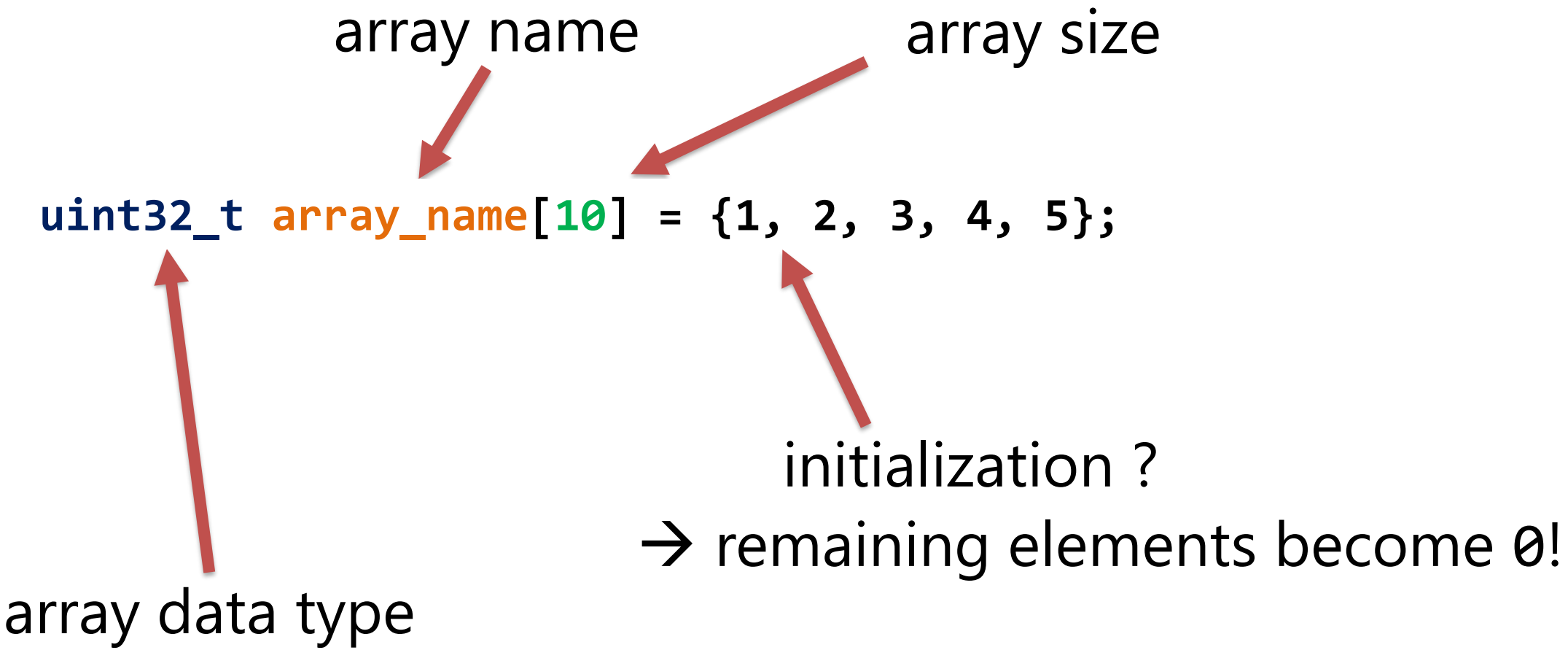
```c
void main() {
  float temp_august [NUM_DAYS];

  uint16_t current_day;

  [...] // code to set current_day and read temperature values

  temp_august[current_day - 1] = /* some temperature value */;

  [...] // further code
}
```

- array: collection of elements of the **same** data type
- elements are stored in **contiguous** memory → constant access time/cost
- arrays have a **fixed** size known at **compile** time
- indexing starts at 0 → last index is `size - 1`
- changing amount elements at a single point in code → use constants!

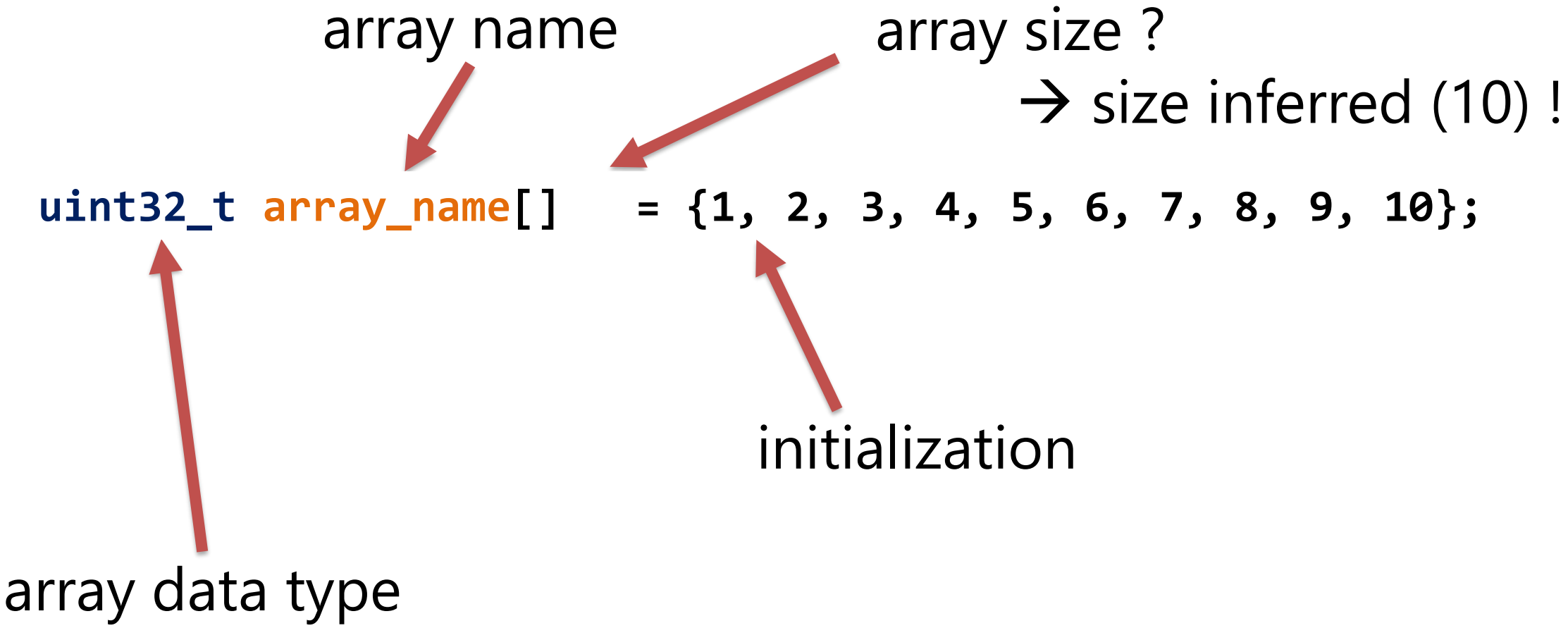
How do I define, declare, and initialize arrays?



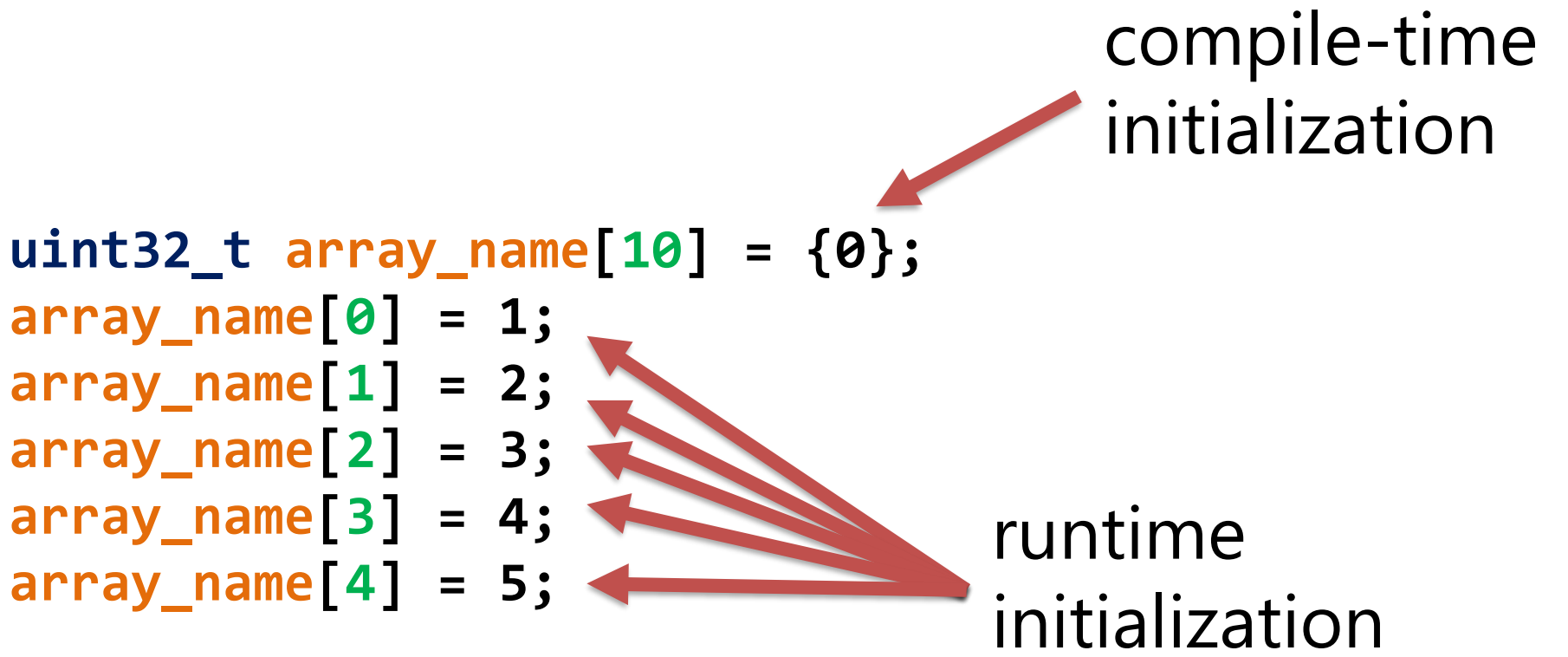
How do I define, declare, and initialize arrays?



How do I define, declare, and initialize arrays?



How do I define, declare, and initialize arrays?



How do I access array elements?

- index operator `[]` for access/assignment of a specific element

```
uint32_t arr[5] = {1, 2, 3, 4, 5};
arr[2] = 12;          // change third element to 12
```

- be careful: C does **not** perform bound checks

```
arr[6] = 12;          // out of bounds!! but legal syntax
```

- use loops to process arrays → mind bounds!

```
uint32_t arr[] = {1, 2, 3, 4, 5, 6};

for(uint32_t i = 0; i < 6; i++) {
  printf("%d\n", arr[i]);
}
```

- use `sizeof(arr)/sizeof(arr[0])` to calculate size (if known at compile time!)

What does contiguous memory mean?

C main.c

```c
uint32_t arr[] = {1, 2, 3, 4, 5, 6};

for(uint32_t i = 0; i < 6; i++) {
  printf("0x%x: %d\n", &arr[i],  arr[i]);
}
```

Output:

```
0x20041fd8: 1
0x20041fdc: 2
0x20041fe0: 3
0x20041fe4: 4
0x20041fe8: 5
0x20041fec: 6
```

C main.c

```c
double arr[] = {1, 2, 3, 4, 5, 6};

for(uint32_t i = 0; i < 6; i++) {
  printf("0x%x: %f\n", &arr[i],  arr[i]);
}
```

Output:

```
0x20041fc0: 1.000000
0x20041fc8: 2.000000
0x20041fd0: 3.000000
0x20041fd8: 4.000000
0x20041fe0: 5.000000
0x20041fe8: 6.000000
```

C main.c

```c
uint8_t arr[] = {1, 2, 3, 4, 5, 6};

for(uint32_t i = 0; i < 6; i++) {
  printf("0x%x: %d\n", &arr[i],  arr[i]);
}
```

Output:

```
0x20041fe8: 1
0x20041fe9: 2
0x20041fea: 3
0x20041feb: 4
0x20041fec: 5
0x20041fed: 6
```

Arrays and functions

- as function parameters → arrays decay to pointers
- function receives the base address (= address of the first element)
- modifying the array in the function affects original array → “call by reference”

```
void process_array(uint32_t arr[], uint8_t n) {
    for (size_t i = 0; i < n; i++) {
      [...]
    }
}


[...]
uint32_t data[8];
[...]
process_array(data, 8);
```

What if one dimension is not enough?

- one-dimensional array → vector
- two-dimensional array → matrix
- n-dimensional array → nth-order tensor

- declaration:

```
uint32_t matrix_name[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

- memory is still contiguous (row-major order) → each row after another

What if one dimension is not enough?

- use nested loops to iterate!

main.c

```c
uint32_t matrix[3][3][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9,
                            10, 11, 12, 13, 14, 15, 16, 17, 18,
                            19, 20, 21, 22, 23, 24, 25, 26, 27};

for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 3; j++) {
    for (int k = 0; k < 3; k++) {
      printf("%d ", matrix[i][j][k]);
    }
    printf("\n");
  }
  printf("\n");
}
```

Output:

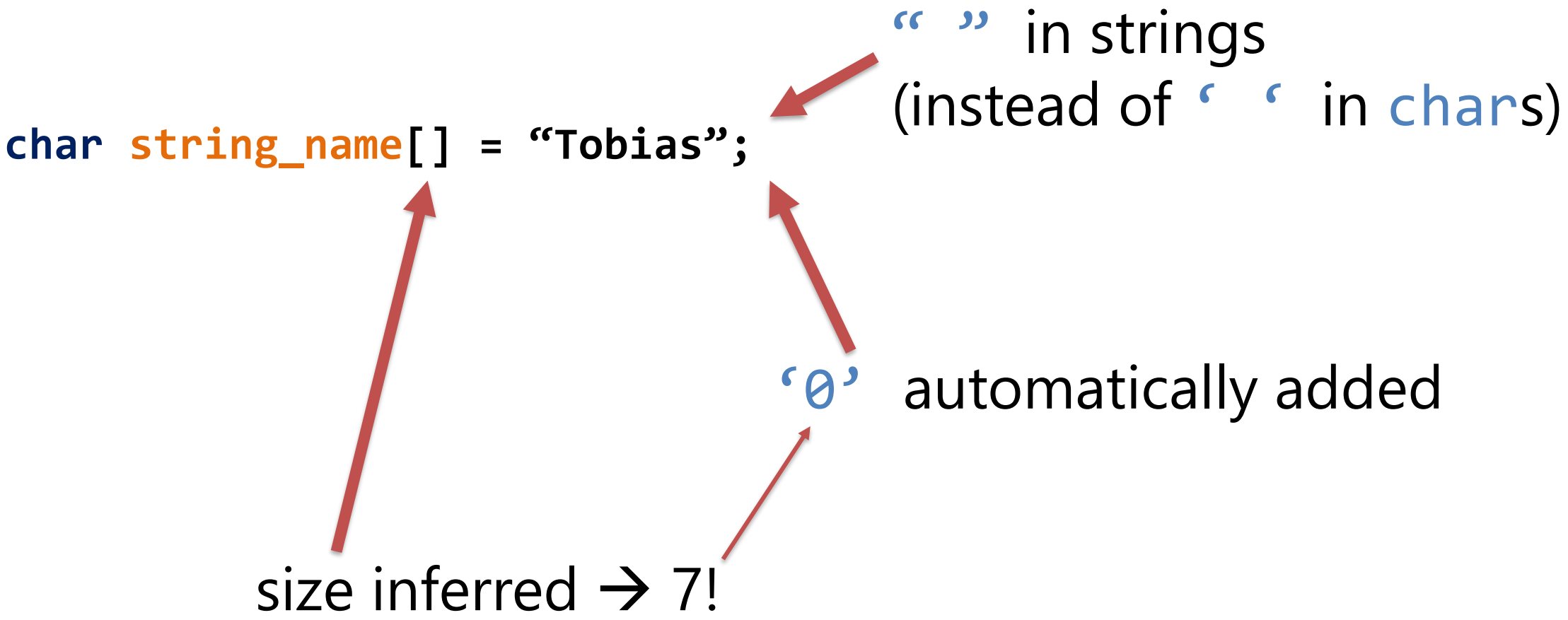
```
1 2 3
4 5 6
7 8 9

10 11 12
13 14 15
16 17 18

19 20 21
22 23 24
25 26 27
```

What is a string? Why do I need strings?

- In C, a string is
  - a sequence of (ASCII) characters,
  - terminated by a null character `'\0'`, and
  - stored in an array of type `char`.

- example:

How do I define, declare, and initialize strings?



Common string functions

- use `%s` in `printf` to print entire strings

```
char building[] = "Hochspannungshalle";
printf("String: %s\n", building);
```

- use standard string functions declared in `<string.h>`

```
size_t strlen(const char *s);                          // returns the number of characters until '\0'.

char *strcpy(char *dst, const char *src);              // copies src to dst; ensure dst is large enough.
char *strncpy(char *dst, const char *src, size_t n);   // copies at most n characters
                                                       // does not always terminate with '\0‘.

char *strcat(char *dst, const char *src);              // concatenates src onto dst.
char *strncat(char *dst, const char *src, size_t n);   // concatenates at most n characters.

int strcmp(const char *s1, const char *s2);            // lexicographically compares two strings;
                                                       // returns <0, 0, or >0.
int strncmp(const char *s1, const char *s2, size_t n); // compares at most n characters.
```

→ always make sure to have **enough memory** for the destination buffer!

Example: counting characters

main.c

```c
#include <stdio.h>
#include "pico/stdlib.h"
#include <string.h>


void main() {
  stdio_init_all();

  char text[] = "Computer Engineering";
  uint32_t length = 0;
  while(text[length++] != '\0');

  printf("Length: %d / %d\n", strlen(text), length - 1);
}
```

Output:

```
Length: 20, 20
```

Example: traversing a string

main.c

```c
#include <stdio.h>
#include "pico/stdlib.h"
#include <string.h>


void main() {
  stdio_init_all();

  char text[] = "Computer Engineering";

  for (uint32_t i = 0; text[i] != '\0'; i++) {
    printf("%c", text[i]);
  }
}
```

Output:

```
Computer Engineering
```

Example: traversing a string

main.c

```c
#include <stdio.h>
#include "pico/stdlib.h"
#include <string.h>


void main() {
  stdio_init_all();

  char text[] = "Computer Engineering";

  for (uint32_t i = 0; i <= strlen(text); i++) {
    printf("%c", text[i]);
  }
}
```

Output:

```
Computer Engineering
```

Arrays of strings

- array of fixed-length strings:

```
char names1[3][10] = {"Thomas", "Sue", "Anna"};
```

- array of pointers to string literals:

```
const char* names2[] = {"Thomas", "Sue", "Anna"};
```

- difference?

```
for (uint32_t i = 0; i < 3; i++) {
  printf("%10s, len: %d\n", names1[i], sizeof(names1[i]));
  printf("%10s, len: %d\n", names2[i], sizeof(names2[i]));
}
```

Output:

```
    Thomas, len: 10
    Thomas, len: 4
       Sue, len: 10
       Sue, len: 4
      Anna, len: 10
      Anna, len: 4
```

Best practices and Common Pitfalls

- always ensure the index is within `[0, size-1]`
- initialize arrays to avoid unpredictable values; do not rely on default zeros
- pass both the array and its length to functions to avoid pointer decay issues
- avoid overflows by writing more data than the array can hold
- choose the correct data type and size to represent your data

- ensure arrays are large enough to hold the text and terminator
- forgetting the null (`'\0'`) terminator → unpredictable behaviour
- using `==` to compare strings → compares addresses, not content
- "buffer overflows" → copying or concatenating without checking buffer size can lead to security vulnerabilities
- avoid modifying string literals → store string literals in `const` pointers

What's missing?

- dynamically-sized arrays and strings (dynamic memory allocation)
- pointers to array elements and strings and their arithmetic

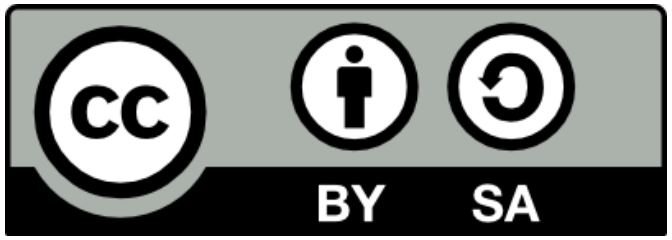
→ will be covered next unit

Practical Examples and Demonstrations

- write a function that takes an integer array and prints the arithmetic mean and the element closest to the mean.
- calculate matrix multiplications
- reverse the elements of an array

- compare strings lexicographically (my own `strcmp`)
- match arrays of names of sensors to their values:
  - suppose we read sensor values and label them by name; combine arrays of floats with arrays of string pointers to make printing more human-readable

27 License



This slide set is licensed under:

CC BY-SA 4.0 International

https://creativecommons.org/licenses/by-sa/4.0/

Tobias Scheipel, TU Graz 2025

https://www.scheipel.com

Notes on Licensing:

- This license applies to all content created by the author and not explicitly labeled as external material.
- External materials in this slide set, such as images or data from third-party sources, retain their respective licenses.
- The slide set uses pictures from https://cocomaterial.com licensed under CC 0.