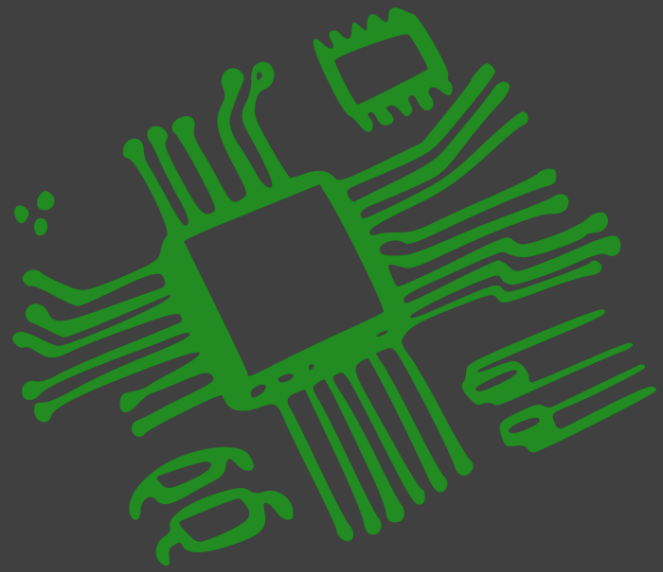
Computer Engineering, VU

(448.012)

— Unit 7: Pointers and Structures —



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

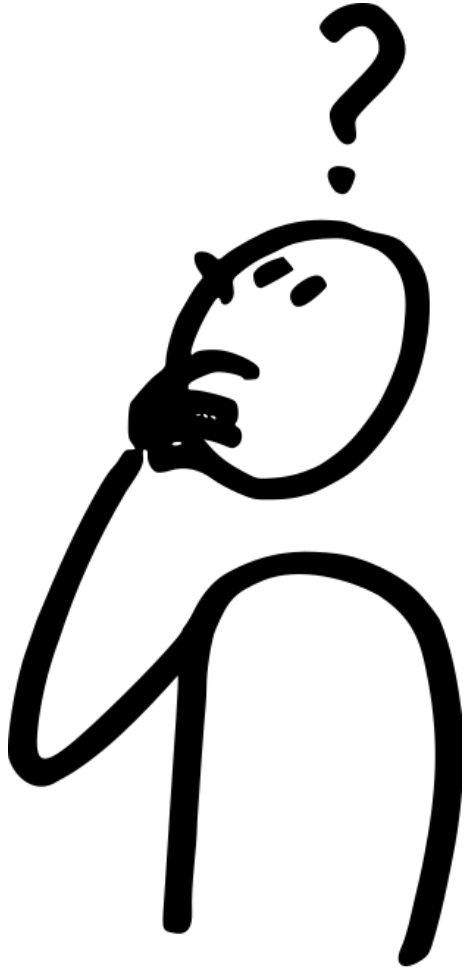
Reconfigurable Computer Architectures
Institute of Technical Informatics
Graz University of Technology

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

General Overview

- What is a pointer? Why do I need pointers?
- Pointer fundamentals and arithmetic
- Pointers and arrays
- Pointers and Functions
- Pointers to pointers and dynamic memory

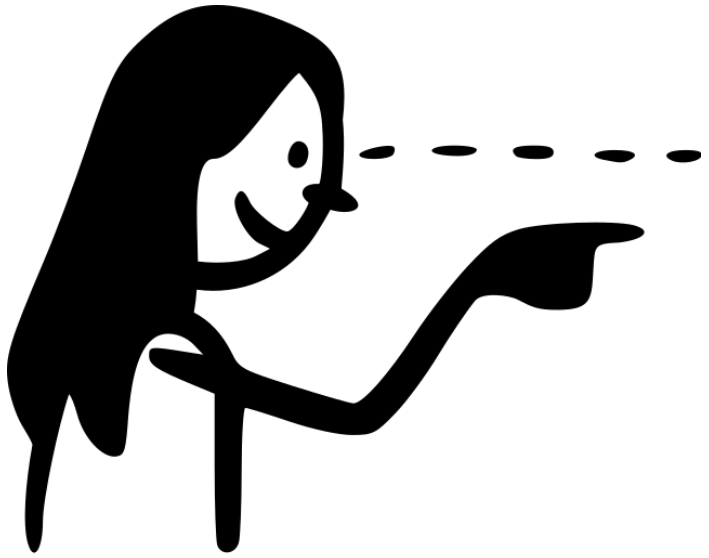
- Enumerations
- Structures
- Unions
- Bitfields



What is a Pointer? Why do I need pointers?

→ fundamental concepts in any C program.

- "a variable that points to another variable"
- A pointer is a variable that holds the address to another variable instead of holding a value directly.



Why?

→ Flexibility!

→ Memory Management!

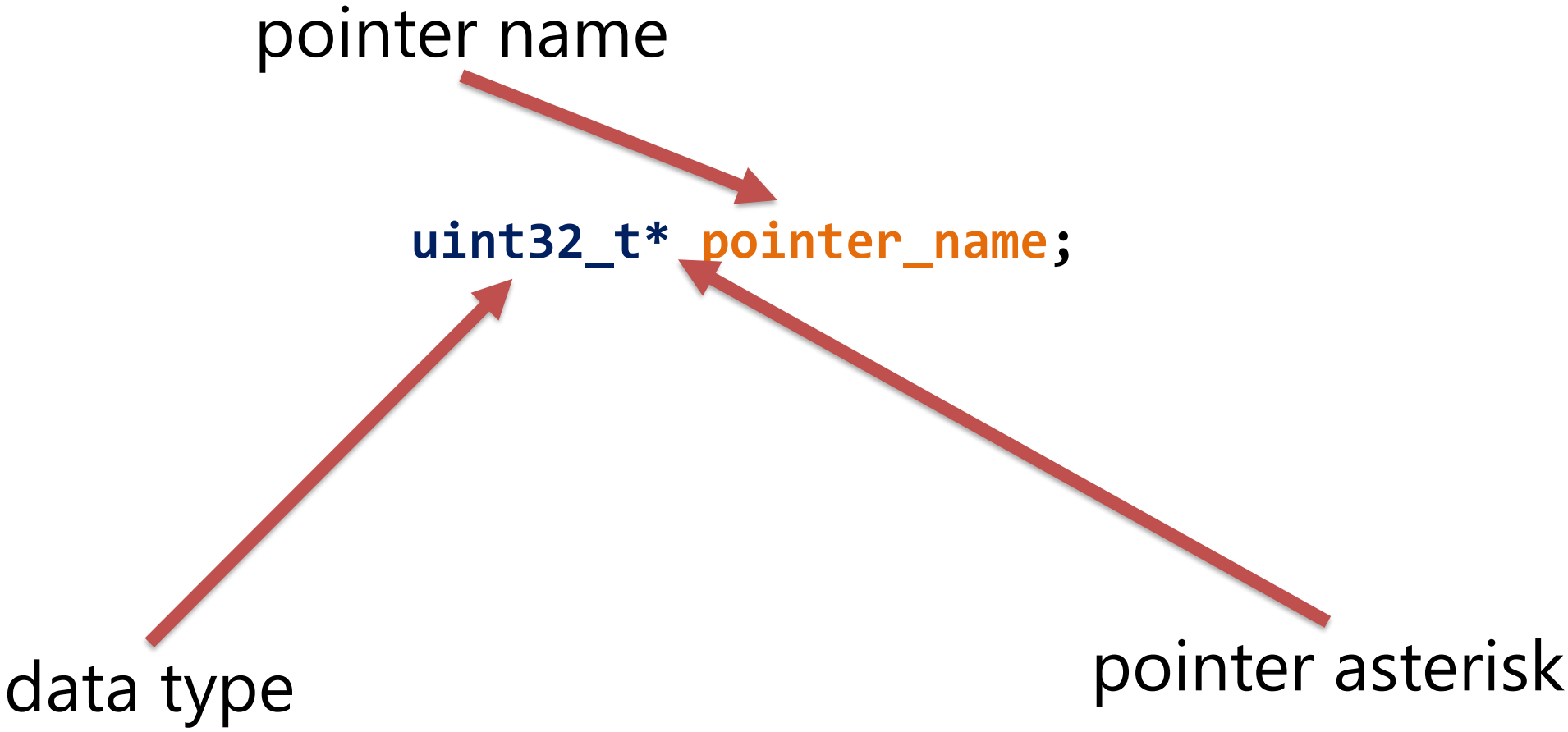
→ Hardware Interfacing!

→ Complex Data Structures!

→ "call by reference"!

→ Control!

What does a pointer variable look like?



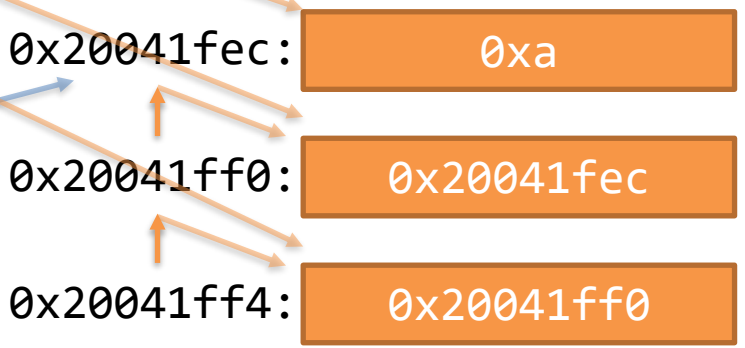
How can I create a pointer?

main.c

```c
uint32_t   var = 0xa;
uint32_t*  ptr = &var;
uint32_t** ptr_ptr = &ptr;

printf("var     = 0x%08x, &var     = 0x%08x\n", var, &var);
printf("ptr     = 0x%08x, &ptr     = 0x%08x\n", ptr, &ptr);
printf("ptr_ptr = 0x%08x, &ptr_ptr = 0x%08x\n", ptr_ptr, &ptr_ptr);
```

Output:

```
var     = 0x0000000a, &var     = 0x20041fec
ptr     = 0x20041fec, &ptr     = 0x20041ff0
ptr_ptr = 0x20041ff0, &ptr_ptr = 0x20041ff4
```



How can I access the value we point at?

main.c

```c
uint32_t   var = 0xa;
uint32_t*  ptr = &var;

printf("var = 0x%08x, &var = 0x%08x\n", var, &var);
printf("ptr = 0x%08x, &ptr = 0x%08x, *ptr = 0x%08x\n", ptr, &ptr, *ptr);

var = 0xc0de;
printf("var = 0x%08x, &var = 0x%08x\n", var, &var);
printf("ptr = 0x%08x, &ptr = 0x%08x, *ptr = 0x%08x\n", ptr, &ptr, *ptr);

*ptr = 0xc0ffee;
printf("var = 0x%08x, &var = 0x%08x\n", var, &var);
printf("ptr = 0x%08x, &ptr = 0x%08x, *ptr = 0x%08x\n", ptr, &ptr, *ptr);
```

Output:

```
var = 0x0000000a, &var = 0x20041fe8
ptr = 0x20041fe8, &ptr = 0x20041fec, *ptr = 0x0000000a
var = 0x0000c0de, &var = 0x20041fe8
ptr = 0x20041fe8, &ptr = 0x20041fec, *ptr = 0x0000c0de
var = 0x00c0ffee, &var = 0x20041fe8
ptr = 0x20041fe8, &ptr = 0x20041fec, *ptr = 0x00c0ffee
```

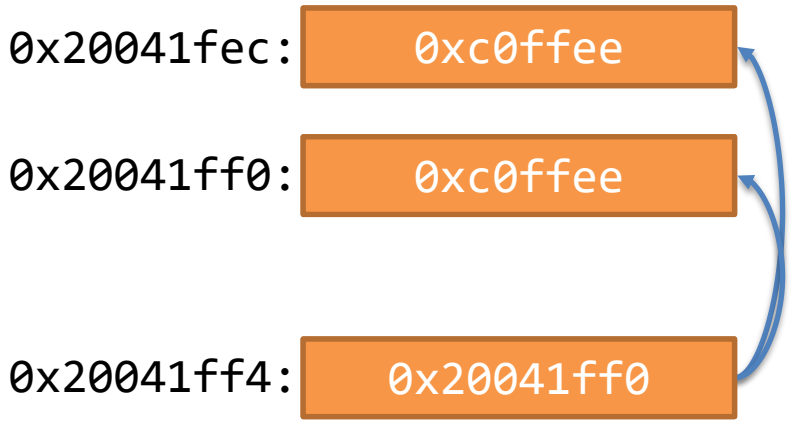
How can I access the value we point at?

main.c

```c
uint32_t   var1 = 0xa, var2 = 0xb;
uint32_t*  ptr = &var1;

printf("var1 = 0x%08x, var2 = 0x%08x\n", var1, var2);

*ptr = 0xc0ffee;
printf("var1 = 0x%08x, var2 = 0x%08x\n", var1, var2);

ptr = &var2;
*ptr = 0xc0ffee;
printf("var1 = 0x%08x, var2 = 0x%08x\n", var1, var2);
```

Output:

```
var1 = 0x0000000a, var2 = 0x0000000b
var1 = 0x00c0ffee, var2 = 0x0000000b
var1 = 0x00c0ffee, var2 = 0x00c0ffee
```



How can I create and access pointers?

Take care!

- Declare a pointer variable with **`datatype* pointer_name;`**
  - Attention: the name is **`pointer_name`**, not **`*pointer_name`**!
  - The pointer points to a value of type **`datatype`**

- Set the value of a pointer with **`&`**-operator (address-of operator)
  - **`pointer_name = &variable;`** sets pointer to address of variable

- Access value at address with **`*`**-operator (dereference operator)
  - **`*pointer_name;`** yields the value stored at the pointed address
  - **`*pointer_name = value;`** sets value at pointing address

How to best initialize a pointer?

```
uint32_t* pointer_name = NULL;
```

- Use a NULL Pointer!
  - → NULL does not point to any valid memory address (usually 0x0)

    for GCC: `#define NULL ((void*)0)`

- Why do I need NULL?
  - → Error Handling
  - → Memory Safety
  - → End Markers

Always **initialize** pointers to NULL!
Always **check** if a pointer is NULL!
Prevent dangling pointers by using NULL!

Pointer Arithmetic

Basic Operations

Pointers can be

- incremented, decremented, and manipulated
- using arithmetic operations.

→ C handles data type size of the pointer automatically!

main.c

```c
uint32_t  var = 42;
uint32_t* var_ptr = &var;

printf("var_ptr = 0x%x, *var_ptr = %d\n", var_ptr, *var_ptr);
var_ptr++;
printf("var_ptr = 0x%x, *var_ptr = %d\n", var_ptr, *var_ptr);
var_ptr++;
printf("var_ptr = 0x%x, *var_ptr = %d\n", var_ptr, *var_ptr);
```

Output:

```
var_ptr = 0x20041ff4, *var_ptr = 42
var_ptr = 0x20041ff8, *var_ptr = 268436068
var_ptr = 0x20041ffc, *var_ptr = 268436003
```

Pointer Arithmetic

Accessing Arrays

- Arrays and pointers are closely related; consider:

```
uint32_t arr[100];
```

- The following lines are identical:

```
arr;
&arr[0];
```

- and so are these:

```
arr[2]
*(arr + 2)
```

main.c

```
1  uint32_t  arr[10] =
2    {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
3
4  uint8_t index = 0;
5  while(arr[index]) {
6      printf("%d, ", arr[index++]);
7  }
8
9  printf("\n");
10
11 uint32_t* iterator = arr;
12 while(*iterator) {
13     printf("%d, ", *iterator++);
14 }
```

Output:

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

Pointers and Functions

Pointers as function parameters

- “call by reference” → see Unit 5
- modifying original data

main.c

```c
// ...

void modifyReference(uint32_t* num) {
    *num = 20;  // Changes the value at the memory address pointed to
}

void main() {
    uint32_t x = 10;
    modifyReference(&x);  // Passing the address of x (call by reference)
    printf("After call by reference: %d\n", x);
}
```

Output:

```
After call by reference: 20
```

Pointers and Functions

Pointers as return values

How can I return an array of values? → especially helpful for dynamic memory allocation

main.c

```c
uint32_t* modifyReference(uint32_t* num) {
  *num = 20;
  *(num + 1) = 23;
  return num; // Returning a pointer
}

void main() {
  uint32_t  arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};

  uint32_t* iterator = modifyReference(arr); // Passing the array

  while(*iterator) printf("%d, ", *iterator++);
}
```

Output:

```
20, 23, 3, 4, 5, 6, 7, 8, 9,
```

Dynamic Memory Allocation

... an introduction

- What if I do not know how much data I need?

→ dynamic memory allocation

- Heap is used instead of stack
- User must maintain heap!

- Advantages:
  - Flexibility: only allocate runtime-required memory
  - Efficiency: avoid large data structures if not needed
  - Complex data structures: dynamic allocation enables runtime-allocation of variable-sized structures

Dynamic Memory Allocation

… how it works

- Allocate memory on the heap

memory size

type cast; NULL on error

```
uint32_t* arr = (uint32_t*) malloc(10 * sizeof(uint32_t));
uint32_t* arr = (uint32_t*) calloc(10,  sizeof(uint32_t));
```

→ uninitialized

→ initialized to 0

- Free your previously allocated memory (mandatory!)

```
free(arr);
```

- Reallocate memory and adjust its size

```
arr = (uint32_t*) realloc(arr, 13 * sizeof(uint32_t));
```

- Useful functions:

```
void* memset(void* dest, int c, size_t n)
void* memcpy(void* dest, const void* src, size_t n)
```

```
void* malloc(size_t size)
void* calloc(size_t n, size_t size)
void free(void* ptr)
void* realloc(void* ptr, size_t new_size)
```

Dynamic Memory Allocation

… why not to use it?

Try to avoid within embedded systems! Here's why:

- limited memory
  → fragmentation
- little/no operating system support
  → management by the developer only
- non-deterministic behaviour
  → violation of deadlines within real-time systems
- memory leaks lead to system panic rather than only a killed task

Dynamic Memory Allocation

… what to use instead!

- Static allocation
  → define all required data structures at compile time
  → memory layout remains known

- Take care of your own dynamic memory
  → create a memory region (e.g., by using a linker script)
  → implement your own allocator logic

- Using only local or static variables

Pointers to Pointers

- Multi-level indirection → pointer that points to another pointer

main.c

```c
uint32_t    a = 10;
uint32_t*   ptr = &a;
uint32_t**  pptr = &ptr;
uint32_t*** ppptr = &pptr;

printf("%d %x\n", a, &a);
printf("%d %x %x\n", *ptr, ptr, &ptr);
printf("%d %x %x %x\n", **pptr, *pptr, pptr, &pptr);
printf("%d %x %x %x %x\n", ***ppptr, **ppptr, *ppptr, ppptr, &ppptr);
```

Output:

```
10 20041fd8
10 20041fd8 20041fdc
10 20041fd8 20041fdc 20041fe0
10 20041fd8 20041fdc 20041fe0 20041fe4
```

Pointers to Pointers

… why would I possibly need this?

- Dynamic memory allocation in general
- Multi-dimensional arrays
- Linked data structures

main.c

```c
typedef struct Node {
  int32_t data;
  struct Node* next;
} Node_t;

void atHead(Node_t** head, int value) {
  Node_t* new_node =
     (Node_t*)malloc(sizeof(Node_t));
  new_node->data = value;
  new_node->next = *head;
  *head = new_node;
}
```

(structs follow in a minute)

main.c

```c
// allocate memory for the rows
uint32_t** array = (uint32_t**) malloc(3 *
                                  sizeof(uint32_t*));
for (int8_t i = 0; i < rows; i++) {
  // allocate memory for each row
  array[i] = (uint32_t*)malloc((i + 1) * sizeof(uint32_t));
}
```

Function Pointers

- A function pointer is a variable that stores the address of a **function** rather than **data**.
- Allows calling functions indirectly through a pointer.

main.c

```c
uint32_t (*func_ptr)(uint32_t, uint32_t);
uint32_t add(uint32_t a, uint32_t b) { return a + b; }

void main() {
    func_ptr = add; // assign the address of 'add' to func_ptr
    printf("%d\n", func_ptr(5, 8)); // call add(5, 8) --> 13
}
```

Function Pointers

… why would I possibly need this?

- Runtime and programmatical function selection

main.c

```c
uint32_t multiply(uint32_t a, uint32_t b) { return a * b; }
uint32_t subtract(uint32_t,uint32_t) { return a - b; }
uint32_t (*ops[2])(uint32_t,uint32_t) = { multiply, subtract };
uint32_t result = ops[0](5, 4); // call multiply(5,4)
```

- Callback functions, event handlers, state machine actions
  → user can provide a function as, e.g., library parameter

- "method"-like behaviour in structs
  → add "functionality" to a data type:

main.c

```c
typedef struct Node {
  int32_t data;
  void (*func)(uint32_t);
  struct Node* next;
} Node_t;
```

What is an Enumeration? Why do I need enums?

- An enumeration (enum) is a user-defined type that assigns names to a set of (enumerated, related) constants.

- Enumerations
  - improve code readability by replacing magic numbers.
  - make code easier to maintain and understand when there are a lot of constants involved.

```
#define NORTH 1
#define EAST  2
#define SOUTH 3
#define WEST  4
```

```
enum {NORTH, EAST, SOUTH, WEST};
```

What does an enum look like? How can I use an enum?

C main.c

```c
#define NORTH 1
#define EAST 2
#define SOUTH 3
#define WEST 4

void main() {
    printf("N: %d, E: %d, S: %d, W: %d\n",
        NORTH, EAST, SOUTH, WEST);
}
```

Output:

```
N: 0, E: 1, S: 2, W: 3
```

C main.c

```c
enum {NORTH, EAST, SOUTH, WEST};


void main() {
    printf("N: %d, E: %d, S: %d, W: %d\n",
        NORTH, EAST, SOUTH, WEST);
}
```

Output:

```
N: 0, E: 1, S: 2, W: 3
```

What does an enum look like? How can I use an enum?

main.c

```c
enum {NORTH, EAST, SOUTH, WEST};


void main() {
    printf("N: %d, E: %d, S: %d, W: %d\n",
        NORTH, EAST, SOUTH, WEST);
}
```

Output:

```
N: 0, E: 1, S: 2, W: 3
```

untagged enum

main.c

```c
enum _DIRECTION_ {NORTH, EAST,
    SOUTH, WEST};

void main() {
    printf("N: %d, E: %d, S: %d, W: %d\n",
        NORTH, EAST, SOUTH, WEST);
}
```

Output:

```
N: 0, E: 1, S: 2, W: 3
```

tagged enum

What does an enum look like? How can I use an enum?

main.c

```c
enum {NORTH = -1, EAST, SOUTH, WEST};


void main() {
    printf("N: %d, E: %d, S: %d, W: %d\n",
        NORTH, EAST, SOUTH, WEST);
}
```

Output:

```
N: -1, E: 0, S: 1, W: 2
```

main.c

```c
enum {NORTH = -1, EAST = 7,
    SOUTH = 9, WEST = -12};

void main() {
    printf("N: %d, E: %d, S: %d, W: %d\n",
        NORTH, EAST, SOUTH, WEST);
}
```

Output:

```
N: -1, E: 7, S: 9, W: -12
```

What does an enum look like? How can I use an enum?

main.c

```c
enum _DIRECTION_ {NORTH, EAST,
    SOUTH, WEST};

void main() {
    printf("N: %d, E: %d, S: %d, W: %d\n",
        NORTH, EAST, SOUTH, WEST);
}
```

Output:

```
N: 0, E: 1, S: 2, W: 3
```

main.c

```c
enum _DIRECTION_ {NORTH = -1, EAST = 7,
    SOUTH = 9, WEST = -12};

void main() {
    printf("N: %d, E: %d, S: %d, W: %d\n",
        NORTH, EAST, SOUTH, WEST);

    enum _DIRECTION_ heading = SOUTH;
    printf("heading: %d\n", heading);
}
```

Output:

```
N: 0, E: 1, S: 2, W: 3
heading: 2
```

tag can be used as data type → but this looks a bit odd

Define your own data type!

→ typedef can be used to create alias names for data types

main.c

```c
typedef unsigned char Byte;
typedef char* String;

void main() {
  Byte my_byte = 42;
  printf("%d\n", my_byte);

  String name = "Frodo";
  printf("%s\n", name);
}
```

existing datatype

new alias

Output:

```
42
Frodo
```

main.c

```c
typedef enum _DIRECTION {NORTH, EAST,
    SOUTH, WEST} Direction;


void main() {
  Direction heading = SOUTH;
  printf("heading: %d\n", heading);
}
```

Output:

```
2
```

→ can be used as “normal” data type from now on

What is a Structure? Why do I need structures?

- A structure (struct) is a user-defined data type that groups different types of variables (possibly of different types) under a single name, allowing you to represent more complex data.

- Structures
  - organize related data into one logical unit, improving code clarity.
  - let you create complex data models that represent real-world entities more naturally than simple arrays or single variables.

```
struct _Hobbit_ {
  char name[30];
  uint16_t year_of_birth;
};
```

What does a structure look like? How can I use a struct?

C main.c

```c
struct _Hobbit_ {
  char name[30];
  uint16_t year_of_birth;
};

void main() {
    struct _Hobbit_ hobbit1 = {"Frodo Baggins", 2968};
    printf("%s (%d)\n", hobbit1.name, hobbit1.year_of_birth);
}
```

Output:

```
Frodo Baggins (2968)
```

What does a structure look like? How can I use a struct?

Define a new type

main.c

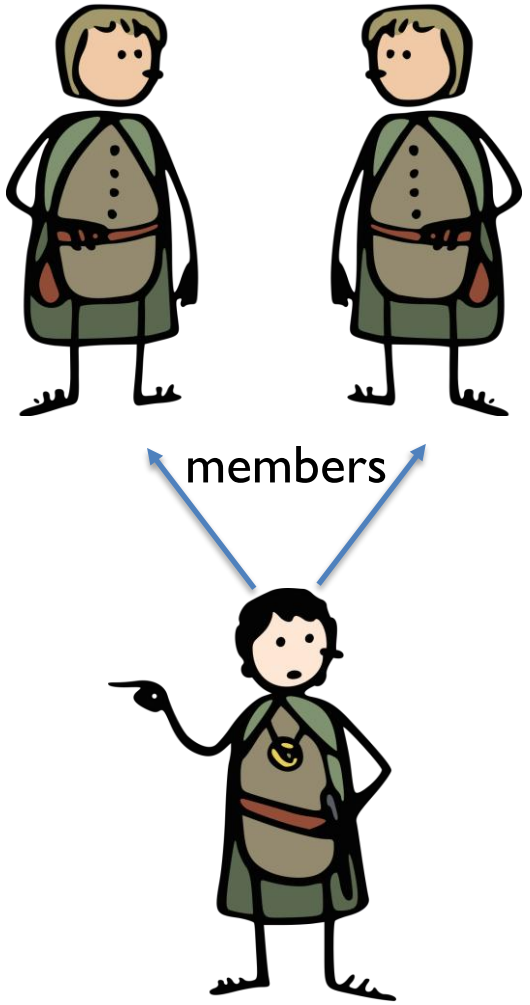
```c
typedef struct _Hobbit_ {
  char name[30];
  uint16_t year_of_birth;
} Hobbit;

void main() {
    Hobbit hobbit1 = {"Frodo Baggins", 2968};
    printf("%s (%d)\n", hobbit1.name, hobbit1.year_of_birth);
}
```

What does a structure look like? How can I use a struct?

Add pointers to the same struct

main.c

```c
typedef struct _Hobbit_ {
  char name[30];
  uint16_t year_of_birth;

  struct _Hobbit_* mother;
  struct _Hobbit_* father;
} Hobbit;

void main() {
    Hobbit hobbit1 = {"Frodo Baggins", 2968, NULL, NULL};
    printf("%s (%d)\n", hobbit1.name, hobbit1.year_of_birth);
}
```



What does a structure look like? How can I use a struct?

Pointers to structures

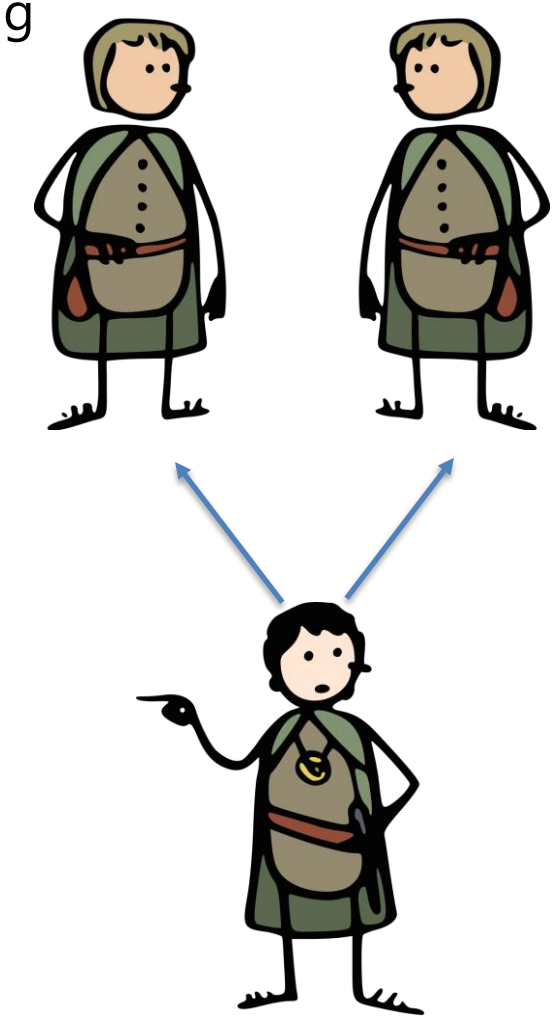
normal member access

dereference and access “normal”

dereference using -> operator

main.c

```c
typedef struct _Hobbit_ {
  char name[30];
  uint16_t year_of_birth;

  struct _Hobbit_* mother;
  struct _Hobbit_* father;
} Hobbit;

void main() {
    Hobbit hobbit1 = {"Frodo Baggins", 2968, NULL, NULL};
    Hobbit* hobbit_ptr = &hobbit1;
    printf("%s (%d)\n", hobbit1.name, hobbit1.year_of_birth);
    printf("%s (%d)\n", (*hobbit_ptr).name,
                        (*hobbit_ptr). year_of_birth);
    printf("%s (%d)\n", hobbit_ptr->name,
                        hobbit_ptr->year_of_birth);
 }
```

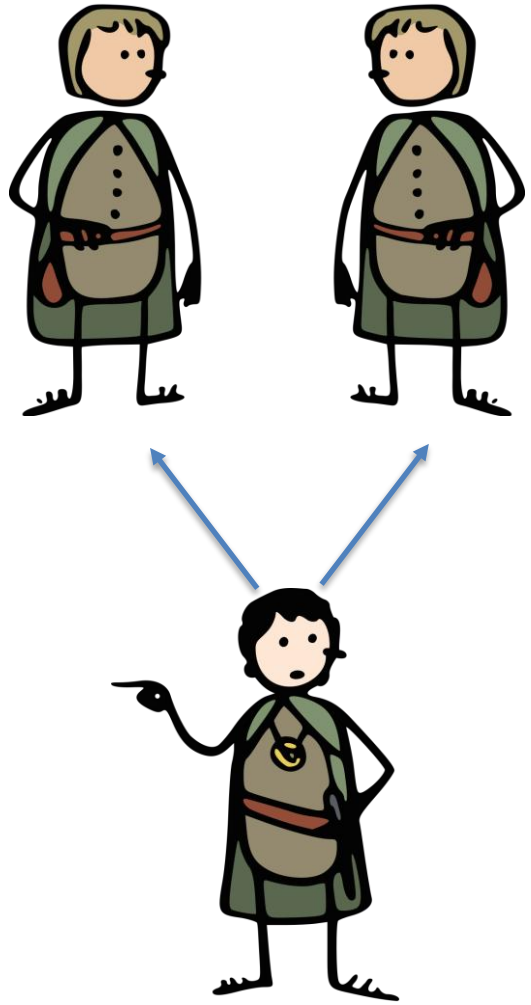


What does a structure look like? How can I use a struct?

Nested structures

main.c

```c
typedef struct _Hobbit_ {
  char name[30];
  uint16_t year_of_birth;

  struct _Hobbit_* mother;
  struct _Hobbit_* father;
} Hobbit;

void main() {
    Hobbit hobbit1 = {"Frodo Baggins", 2968, NULL, NUL

    printf("%s (%d)\n", hobbit1.mother->name,
                        hobbit1.mother->year_of_birth);
}
```



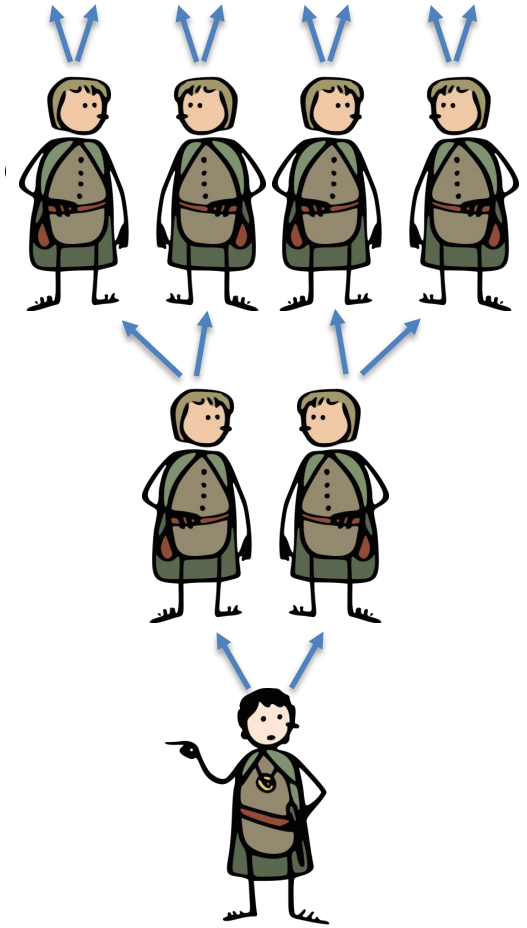
What does a structure look like? How can I use a struct?

Nested structures

main.c

```c
typedef struct _Hobbit_ {
  char name[30];
  uint16_t year_of_birth;

  struct _Hobbit_* mother;
  struct _Hobbit_* father;
} Hobbit;

void main() {
    Hobbit hobbit1 = {"Frodo Baggins", 2968, NULL, NULL};
    Hobbit hobbit2 = {"Primula Brandybuck", 2920, NULL, NULL};
    hobbit1.mother = &hobbit2;
    printf("%s (%d)\n", hobbit1.mother->name,
                         hobbit1.mother->year_of_birth);
}
```

Can be used to create linked data structures (e.g., linked lists, trees)



Output:

```
Primula Brandybuck (2920)
```

What is a Union? Why do I need unions?

- A union is similar to a structure, but all its members overlap and share the same memory location, i.e., having the same starting address. Thus, only one of the members can be used at a time.

- Unions
  - save memory by overlapping data.
  - can be useful when it is already known that a variable will hold different data types at different times.

```
union _SensorData_ {
    uint16_t value;
    uint8_t  bytes[2];
};
```

```
typedef union {
    uint16_t value;
    uint8_t  bytes[2];
} SensorData;
```

What does a union look like? How can I use a union?

```
typedef union {
    uint16_t value;
    uint8_t  bytes[2];
} SensorData;

SensorData data;
SensorData* ptr = &data;
```

- access syntax the same as in structs:

```
data.bytes[0] = 0x1A;
data.value = 0xC0DE;
```

```
ptr->bytes[1] = 0x2B;
ptr->value = 0xBEEF;
```

- easy data re-interpretation:

```
data.bytes[0] = 0xC0;
data.bytes[1] = 0xDE;

printf("%X\n", data.value); // C0DE?
```

Output:

```
DEC0
```

→ consider endianness

- union is always as big as the biggest member!

```
typedef union {
    uint16_t value;
    uint8_t  bytes[8];
} SensorData;
```

What is a Bitfield? Why do I need bitfields?

- A bitfield enables a more precise specification of the bit lengths within a structure.

- Bitfields
  - facilitate fine-grained control over memory usage.
  - can be useful for representing small numeric values like flags or hardware register fields.

```
typedef struct _Status_ {
    uint8_t flag1 : 1;
    uint8_t flag2 : 1;
    uint8_t errorCode : 6;
} Status;
```

  - Bitfields can be accessed as structs.

Best practices and Common Pitfalls

- **Initialize Pointers Properly:** Always initialize pointers to NULL or a valid memory address.
- **Check for NULL before Dereferencing:** Always check if a pointer is NULL before dereferencing it.
- **Avoid Pointer Arithmetic for Unknown Types:** Pointer arithmetic should be done carefully. Ensure the pointer is pointing to the correct type and that the type size is correctly handled.
- **Use Function Pointers Wisely:** When using function pointers, ensure that the pointer points to a valid function before calling it.
- **Keep Structs Compact and typedef them:** Align struct members in a way that minimizes unused space. Use typedef to give a struct a more user-friendly name.

- **Common Pitfalls:**
  - Dereferencing NULL pointers
  - Memory leaks with dynamic memory allocation
  - Out of bounds access
  - Pointer arithmetic errors
  - Dangling pointers
  - Memory alignment issues in own types
  - Mix up structures and unions
  - Bitfield misalignments
  - Unintended enum value assignment
  - Passing large structs by value

Practical Examples and Demonstrations

- Basic Pointer Operations
  - Declare a variable and a pointer that points to the variable's memory address.
  - Dereference the pointer to access the value stored at the address.
  - Modify the value of the variable through the pointer by dereferencing it.
- Dynamic Memory Allocation
  - Allocate memory dynamically for a variable or array using malloc.
  - Assign values to the dynamically allocated memory.
  - Print the values stored in the dynamically allocated memory.
  - Free the dynamically allocated memory to avoid memory leaks.
- Using Function Pointers
  - Define functions and declare a function pointer that can store the address of a function.
  - Assign the address of a function to the function pointer.
  - Call the function using the function pointer, passing the necessary arguments.

Practical Examples and Demonstrations

cont'd.

- Pointer to Structures
  - Define a structure with relevant members.
  - Declare a variable of the structure type and initialize it.
  - Create a pointer to the structure and point it to the structure variable.
  - Access the structure's members through the pointer using the appropriate operator.
- Enums and Pointers
  - Define an enum with named constants.
  - Declare a pointer to an enum and assign it a valid enum value.
  - Access the value of the enum through the pointer.
- Unions
  - Define a union with different members that share the same memory location.
  - Declare a union variable and assign a value to one of the union's members.
  - Access the members of the union, keeping in mind the current "valid value".

Practical Examples and Demonstrations

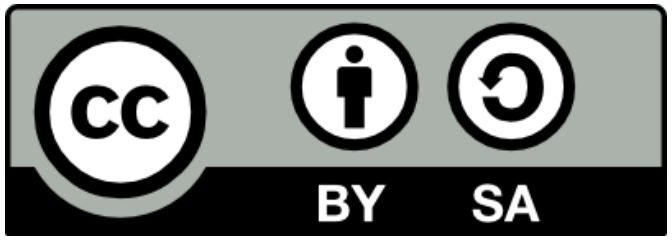
cont'd.

- Bitfields
  - Define a structure with bitfield members, specifying the number of bits for each member.
  - Initialize the bitfield structure with values for the members.
  - Access and modify the bitfield members, ensuring the values fit within the specified bit limits.

What's next?

- Today: Coding Session
- Next Week:
  - Homework 2 will be announced
  - Coding Session
- future Lectures (next Year):
  - on Advanced Embedded Concepts (not relevant for homework)

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.