INF1004 procedural programming in C

DHBW Stuttgart
Christian Holz
christian.holz@lehre.dhbw-stuttgart.de

Lecture 02

PART I

- Recap of the previous contents
- Dynamic Memory Management
- Single Linked Lists
- Double Linked Lists

PART II

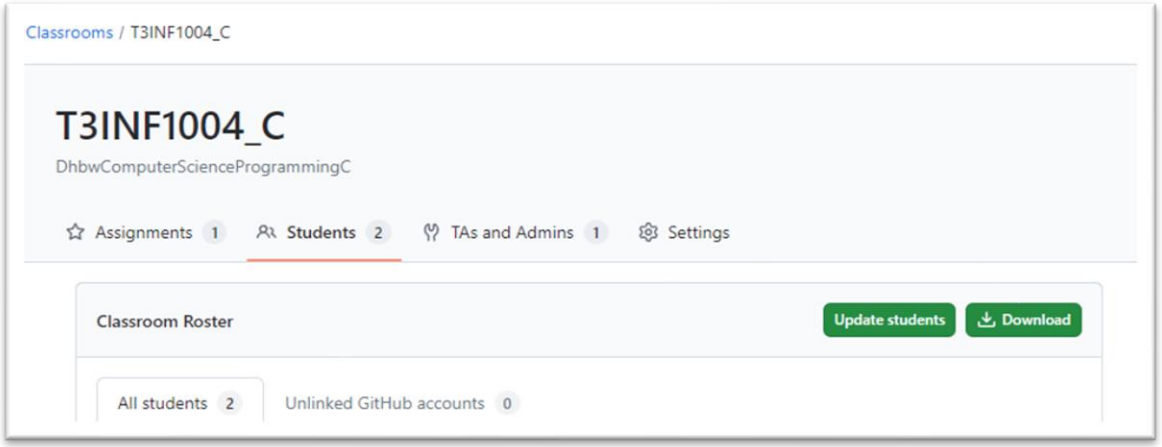
- Programming Exercises I and II
- Parameter At Program Start
- Testing

Recap of the previous contents

Live demo

Your Classroom for C coding Assignments

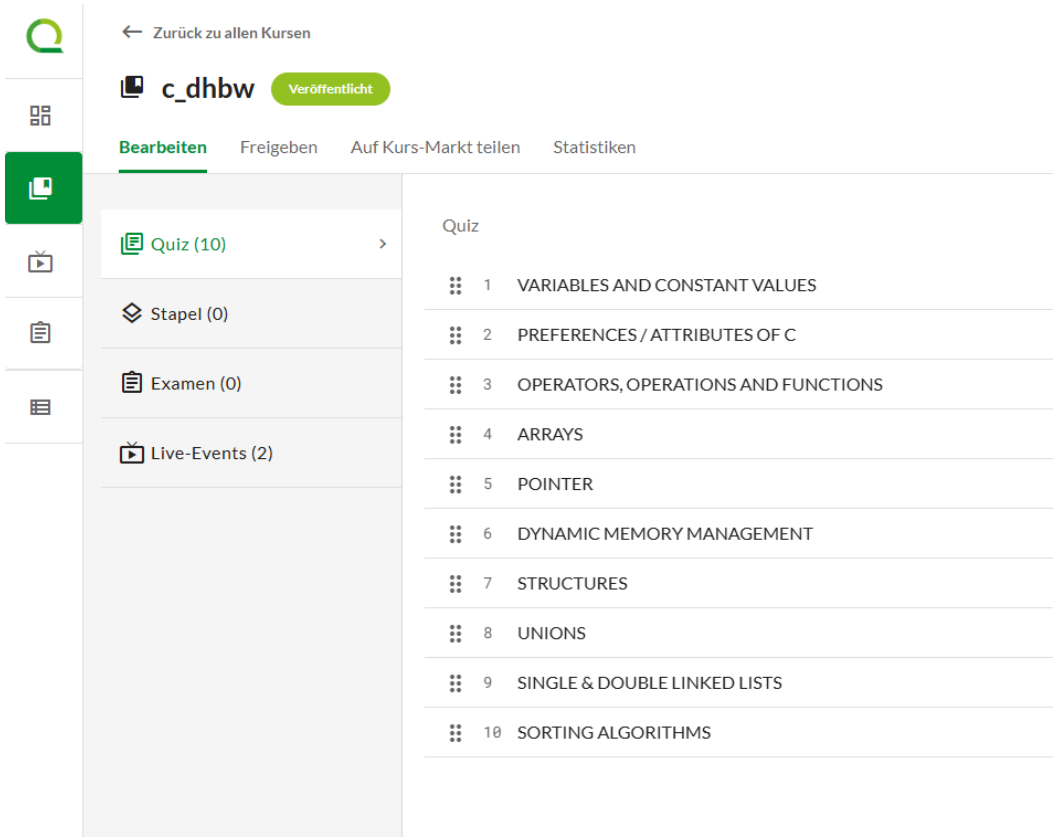
- Let's come together in the Classroom
- **01-Assignment** will be available for you
- Get the Repository
- **01-Assignment** Q&A



https://classroom.github.com/classrooms/182848101-t3inf1004_c/roster

Live event

Let’s play



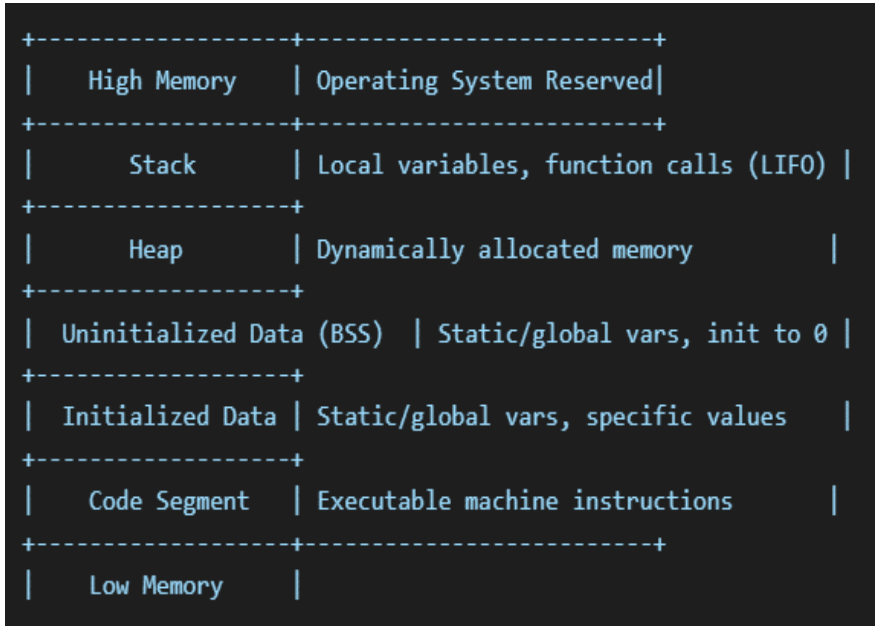
Dynamic Memory Management

Why do we need Memory Management?

Dynamic Memory Management
Memory Layout Regions

- When a C program runs, the memory is divided into different regions, each serving a specific purpose.
- Understanding the memory layout is critical for effective programming and debugging.

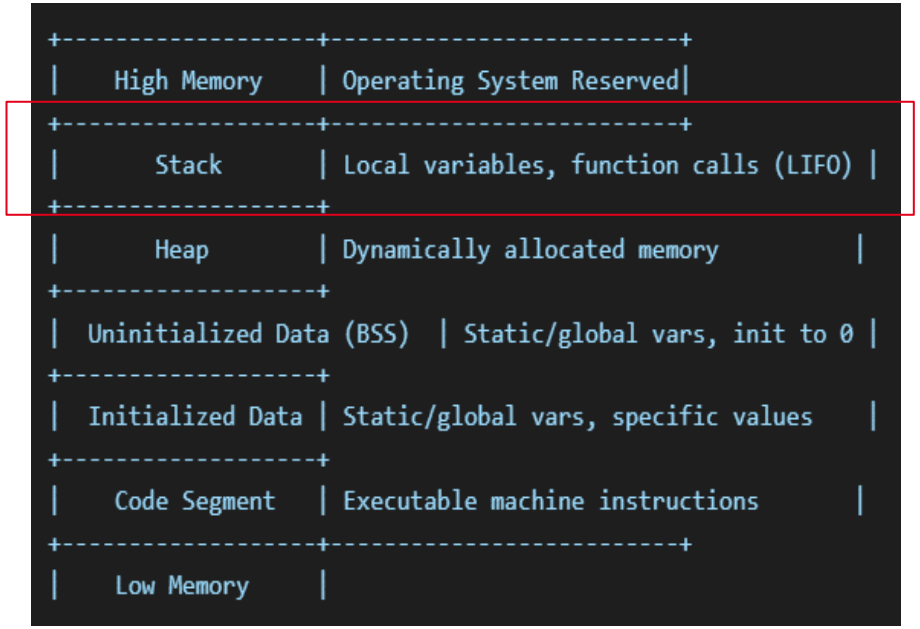
1. Code Segment
2. Data Segment
3. Stack
4. Heap



Dynamic Memory Management
Memory Layout Regions

Stack memory

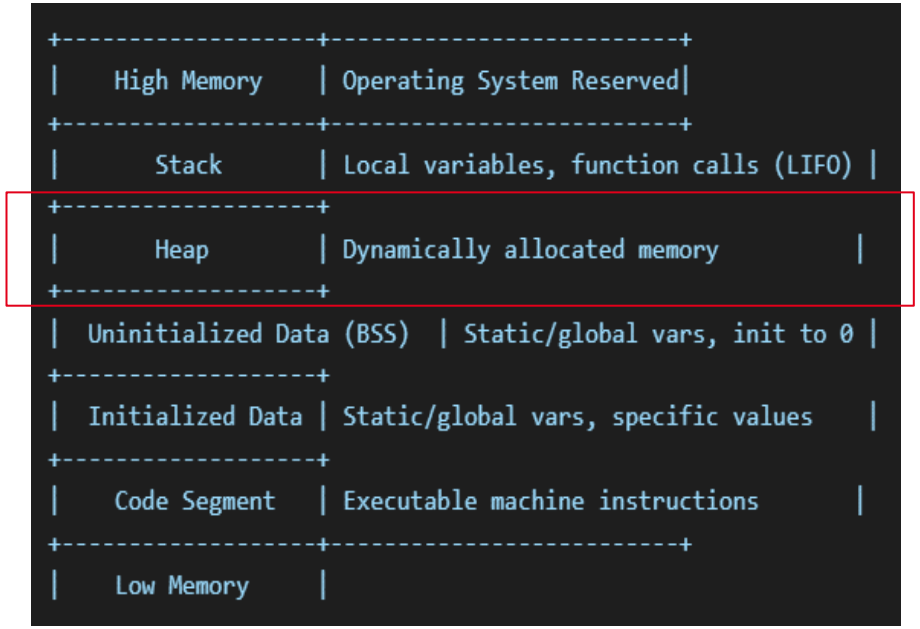
- Stack is a Last-In-First-Out (LIFO) data structure
- Can be controlled very quickly and easily by the operating system
- Programme variables are stored in the stack by default (static, **no changes possible at runtime**)
- Limited to a few MB (~10MB)
- Memory release on the stack is controlled by the operating system



Dynamic Memory Management
Memory Layout Regions

Heap memory

- No memory limitation by the programme (theoretically the entire RAM could be used)
- Variable **sizes can be changed dynamically**
- Variables can be created globally
- Programmer must take over memory management (dynamic memory management in C via functions)

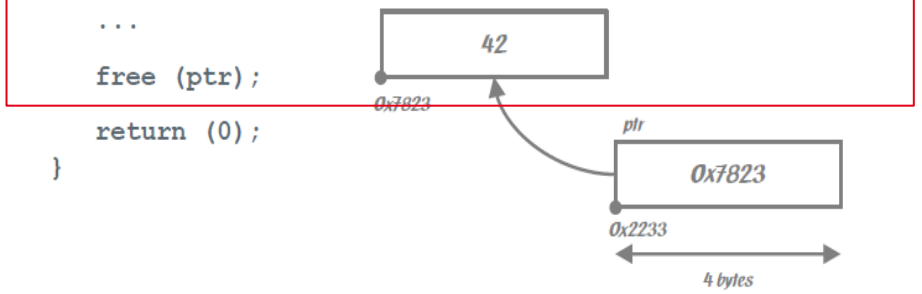


Dynamic Memory Management
Pointers: Dynamic Memory Allocation

- With the dynamic memory management memory space can be allocated during runtime of a program.
- Different to variables, the dynamically allocated memory space does not have a name.
- Therefore, a pointer must point to that area to have access to it
- If the address is lost, the space is lost.

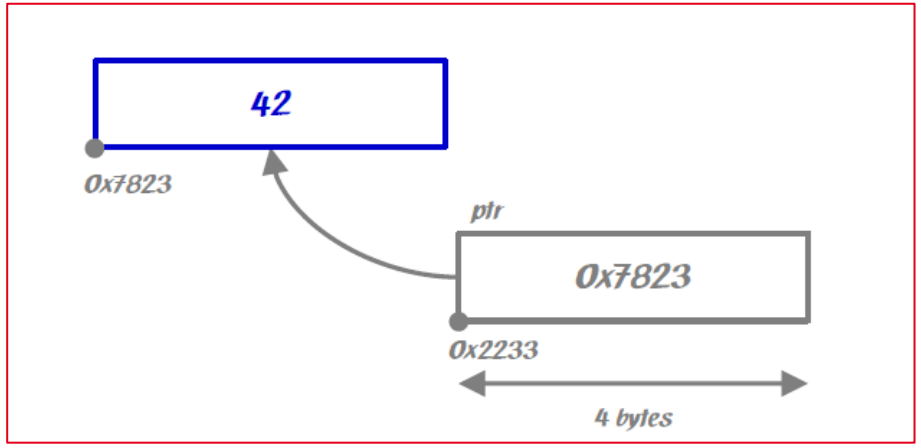
There is no garbage collector available, like in other programming languages.

```
#include <stdio.h>

int main (void)
{
   int * ptr;

   ptr = (int *) malloc (sizeof(int));

   * ptr = 42;

   ...

   free (ptr);

   return (0);
}
```

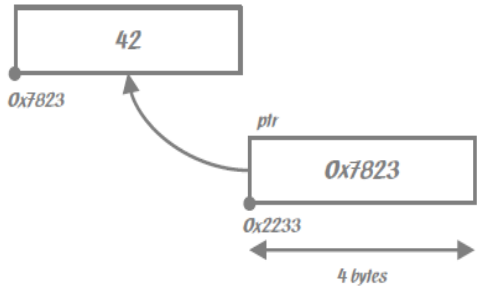


Dynamic Memory Management
Pointers: Dynamic Memory Allocation

- The allocated memory space at 0x7823 does not have an identifier (name).
- The allocated memory space can only be accessed using the address (pointer variable).




```
#include <stdio.h>

int main (void)
{
   int * ptr;

   ptr = (int *) malloc (sizeof(int));

   * ptr = 42;

   ...

   free (ptr);

   return (0);
}
```



Dynamic Memory Management
Functions for DMM

https://en.cppreference.com/w/c/memory

```
void * calloc (size_t count, size_t size)
```

The function `calloc ()` reserves `count` times `size` bytes of memory space and returns the address to this reserved memory area. `calloc ()` initializes the memory with NULL bytes.

If the return value is NULL, the memory allocation did not complete successfully.

```
void * malloc (size_t size)
```

The function `malloc ()` reserves `size` bytes of memory space and returns the address to this reserved memory area. The memory are is not initializes with any pre-defined value.

The return value is the same as with `calloc()`.

Dynamic Memory Management
Functions for DMM

https://en.cppreference.com/w/c/memory

```
void * realloc (void * ptr, size_t size)
```

`Realloc ()` changes the size of the allocated memory space connected with the address `ptr` up to or down to `size` bytes. The content stays unchanged unless the area is decreases.

If decreasing, the content stays unchanged within the area left. If decreasing, space is added but uninitialized.

In case `ptr==NULL`, the function `realloc()` has the same effect as `malloc()`. In case `ptr!=NULL`, the function frees the memory and has the same effect as `free()`.

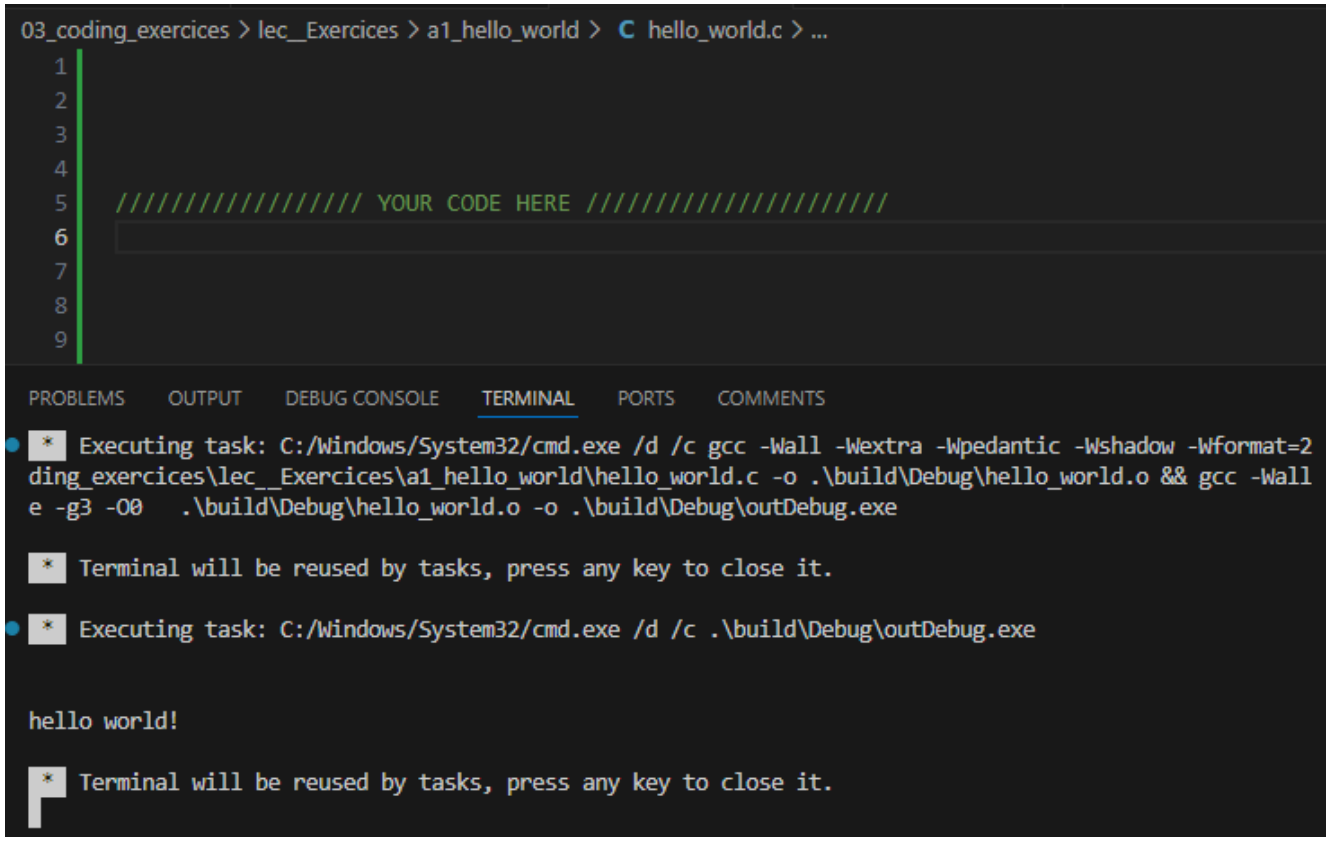
Information: All memory allocated with `malloc()`, `calloc()` or `realloc()` has the capability to host any object.

```
void free (void * ptr)
```

The memory area connected to `ptr` is being freed by `free ()`.

Information: You cannot achieve the same result with `ptr=NULL`. The memory area must be created by `calloc()`, `malloc ()` or `realloc ()`.

Let's code


03_coding_exercices > lec__Exercices > a1_hello_world > C hello_world.c > ...
1
2
3
4
5 ////////////////// YOUR CODE HERE //////////////////////
6
7
8
9
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
* Executing task: C:/Windows/System32/cmd.exe /d /c gcc -Wall -Wextra -Wpedantic -Wshadow -Wformat=2 ding_exercices\lec__Exercices\a1_hello_world\hello_world.c -o .\build\Debug\hello_world.o && gcc -Wall e -g3 -O0 .\build\Debug\hello_world.o -o .\build\Debug\outDebug.exe
* Terminal will be reused by tasks, press any key to close it.
* Executing task: C:/Windows/System32/cmd.exe /d /c .\build\Debug\outDebug.exe
hello world!
* Terminal will be reused by tasks, press any key to close it.

Single Linked Lists

Why?

Single Linked Lists
Intro

A linked list is a data structure in which elements (nodes) are dynamically linked to each other.

In contrast to arrays:

- The size is flexible (no fixed number of elements required).
- Inserting or removing elements is efficient as no memory needs to be moved.

Example applications: Implementation of queues, stacks or memory-intensive operations.

Single Linked Lists
Intro

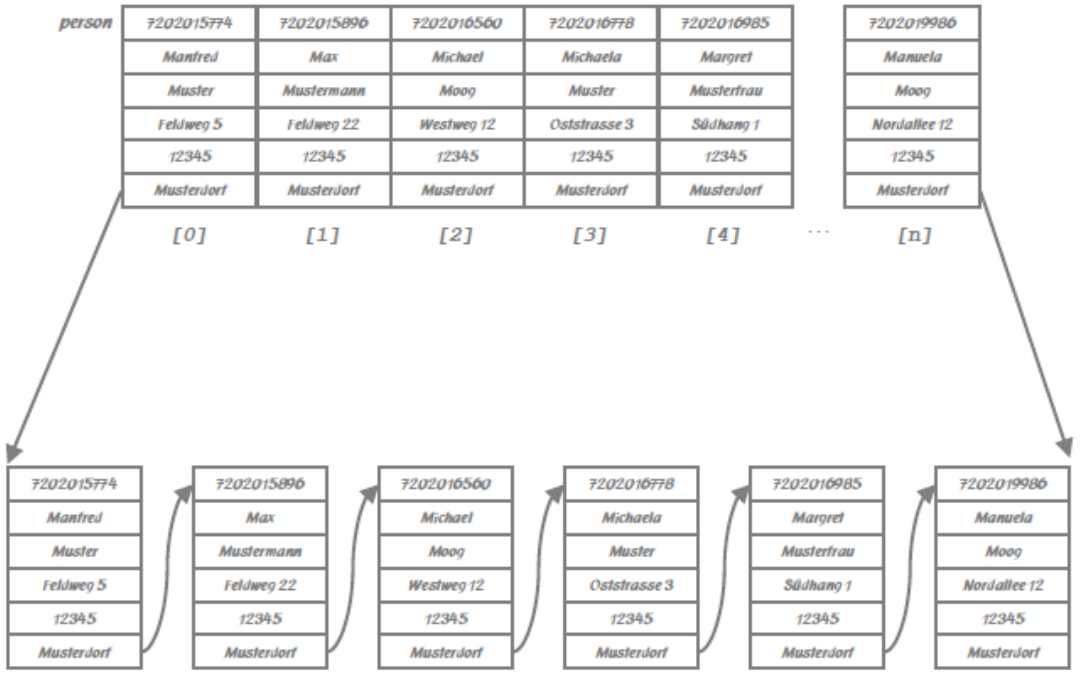
Storing data set in a **array** required to know **how many** data sets must be stored.

An array has a **fixed amount of elements** and cannot change its dimension while the program is executed.

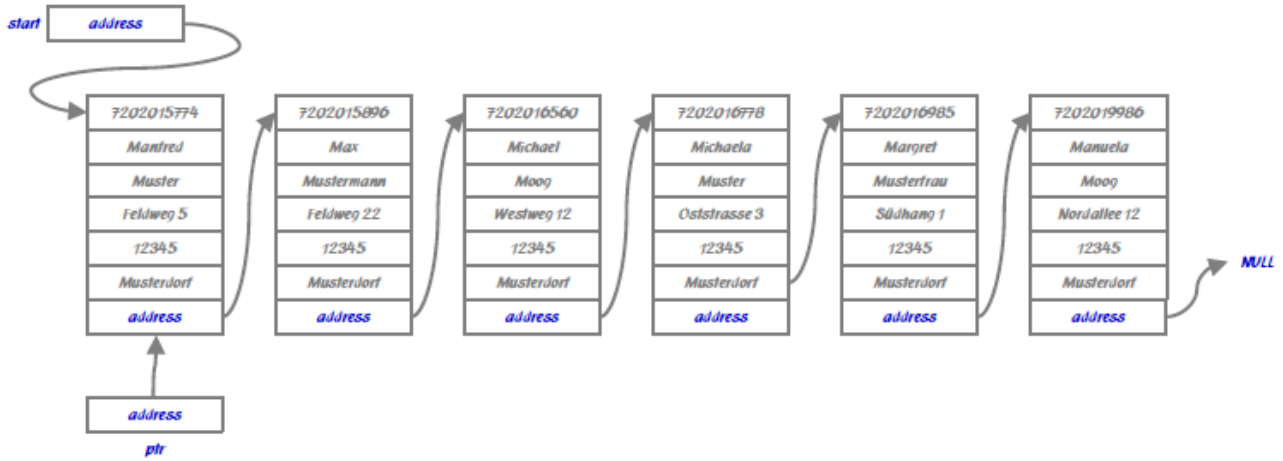
Dynamic data structures offer a higher grade of **flexibility** when storing data:

- trees
- lists

Whilst a static variable has an identifier (name) a static lifetime, the dynamic data structure have a **dynamic lifetime** and **no identifier**.



Single Linked Lists
Intro



An **additional memory** area is needed to store the address to the next element.

In a **single linked list** the address of the **next element** is stored. The last element must point to NULL to indicate the end of the list.

To keep the first element available a **pointer** is mandatory that constantly point **to the first element**.

Access to the elements of the list is an **additional pointer** which point to the single elements of the list.

Information:

All static variables have allocated memory space on the stack. The dynamic data structures will use memory on the heap.

Stack and heap are different memory areas.

Single Linked Lists
Creating a single linked list example

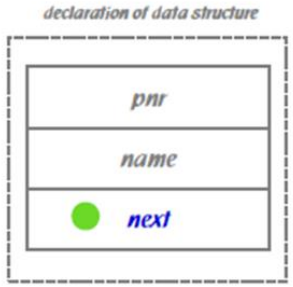
- Creating a single linked list requires an address to the structure itself.
- The pointer **next** is part of the structure.

  The function **malloc()** support to get memory space allocated.
- **malloc() returns the address** to the allocated memory area.

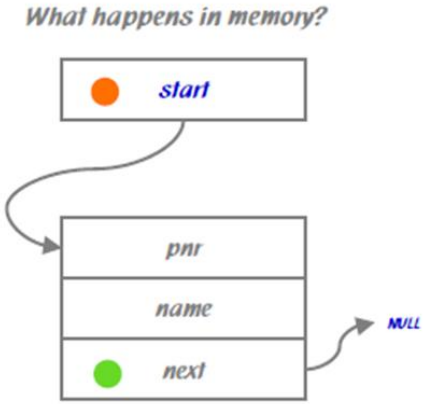
  malloc() return an address of type **void**. The pointer **start** is of type **struct list**.

  This needs to be adapted! (typecast)

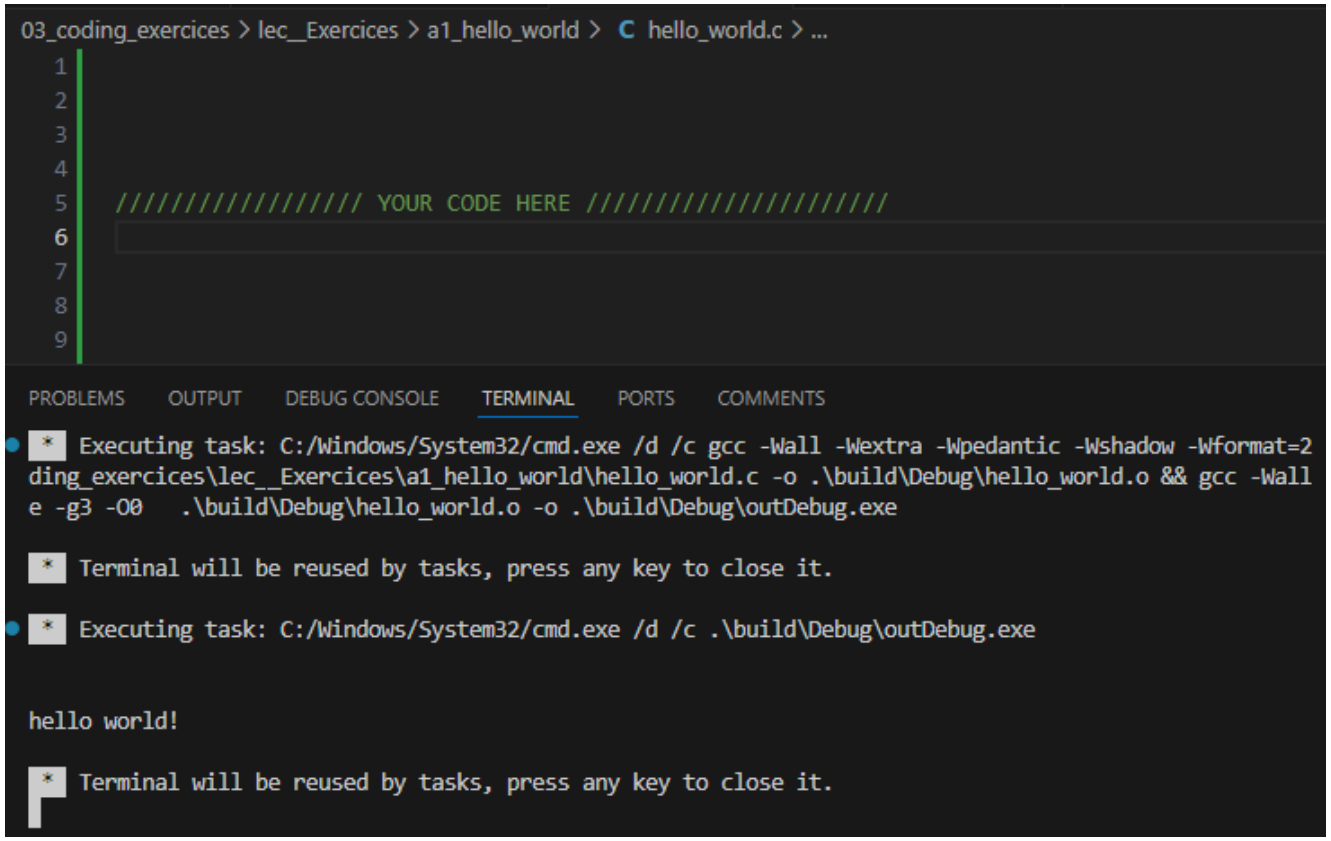
```
struct list {
   int pnr;
   char name[20+1];
   struct list * next;
};
```




```
struct list * start = NULL;

start = (struct list *) malloc (sizeof(struct list));
start->next = NULL;
```



Let's code


03_coding_exercices > lec__Exercices > a1_hello_world > C hello_world.c > ...
1
2
3
4
5 ////////////////// YOUR CODE HERE //////////////////////
6
7
8
9
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
* Executing task: C:/Windows/System32/cmd.exe /d /c gcc -Wall -Wextra -Wpedantic -Wshadow -Wformat=2
ding_exercices\lec__Exercices\a1_hello_world\hello_world.c -o .\build\Debug\hello_world.o && gcc -Wall
e -g3 -O0 .\build\Debug\hello_world.o -o .\build\Debug\outDebug.exe
* Terminal will be reused by tasks, press any key to close it.
* Executing task: C:/Windows/System32/cmd.exe /d /c .\build\Debug\outDebug.exe
hello world!
* Terminal will be reused by tasks, press any key to close it.

Double Linked Lists

Frome single to double?

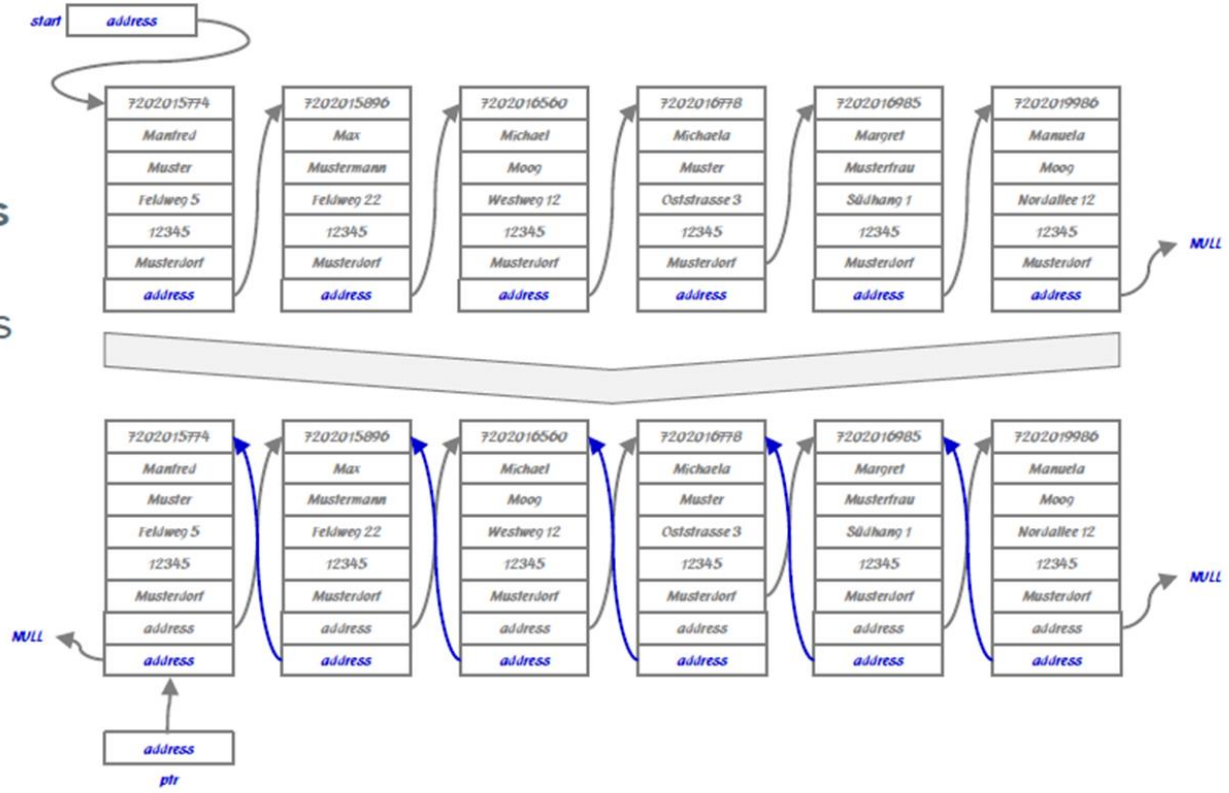
Double Linked Lists
Intro

A **double linked** list has additional pointer included pointing to the **previous element** of the list.

The pointer `prev` of the first element has to point to `NULL` to indicate the start is the linked list.

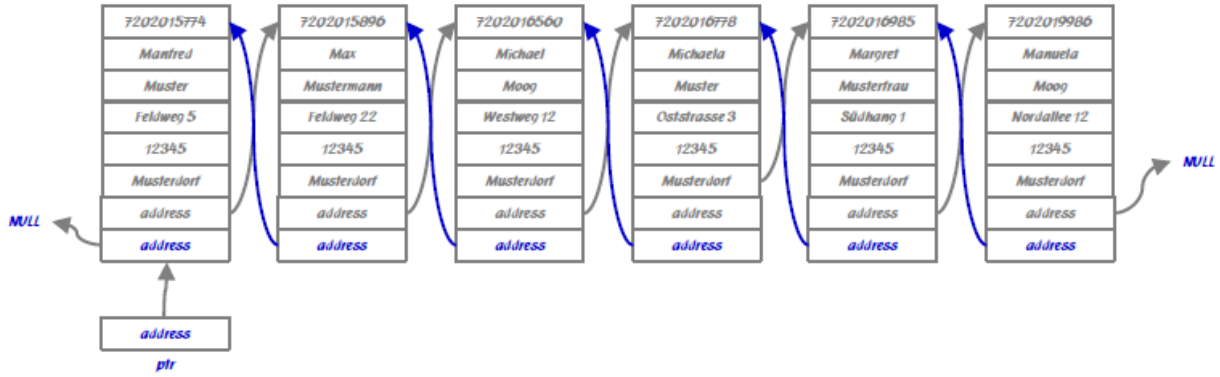
A pointer `ptr` to run through the data structure is still needed.

As long as one pointer points to a list element it is no longer mandatory to maintain the `start` pointer.



Double Linked Lists
Intro

```
struct list {
   int pnr;
   char name[20+1];
   struct list * prev;
   struct list * next;
};

struct list * ptr = NULL;

ptr = (struct list *) malloc (sizeof(struct list));
ptr->next = NULL;
ptr->prev = NULL;
```



Double Linked Lists
Intro

Linked Lists – Code Examples

Create a new element of a double linked list.

```
if (ptr)
{
   printf ("A list already exists.\n");
}
else
{
   ptr = (struct list *) malloc (sizeof (struct list));
   ptr->prev = NULL;
   ptr->next = NULL;
}
```

Hint: If you add a new element to the list, put it at the end.
This approach is the simplest way to do so.

Run through the list using until the end (**forward**)

```
while (ptr->next) ptr=ptr->next;
```

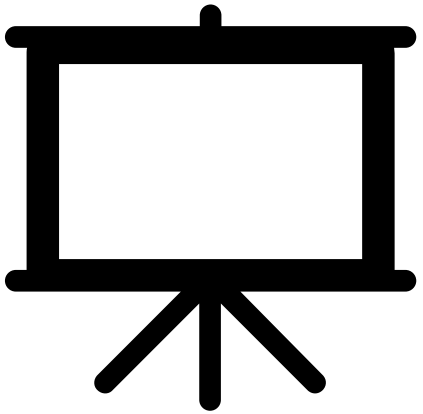
Run through the list using until the start (**backward**)

```
while (ptr->prev) ptr=ptr->prev;
```

Double Linked Lists
Intro

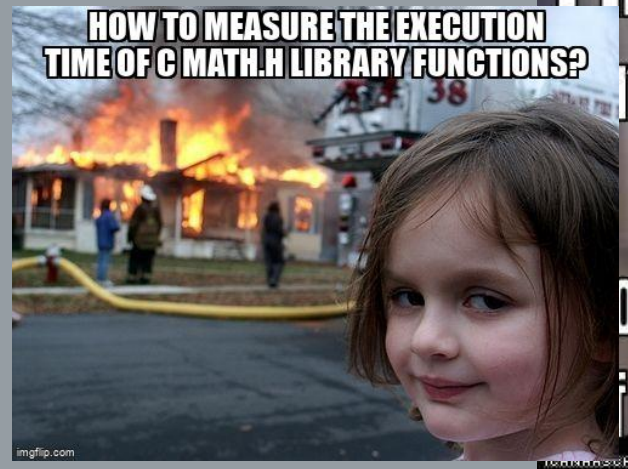
Exercise

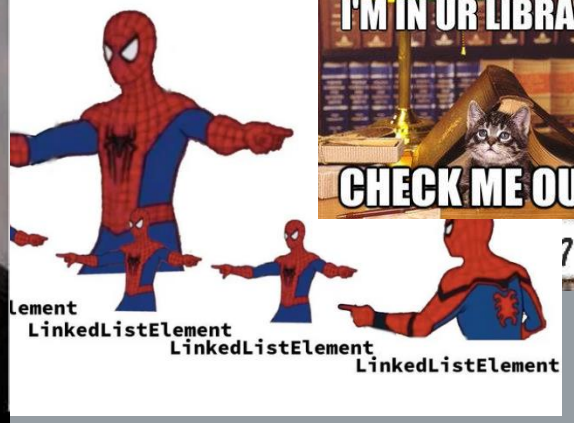
- Work in small groups and start talking about linked lists, dicuss the pro and cons and collect them (linked lists vs. arrays, …)
  - Advantages
  - Disadvantages
- Present your collection

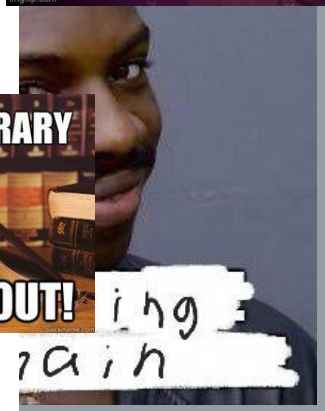


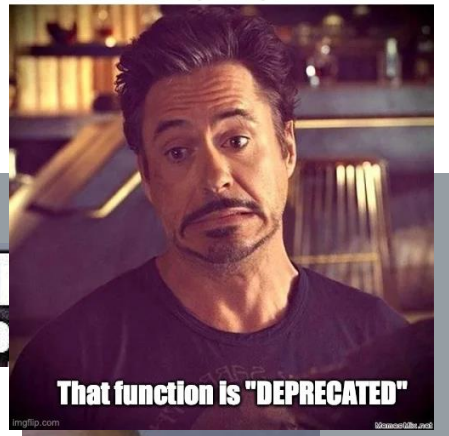
Programming Exercises I and II

Code code code …



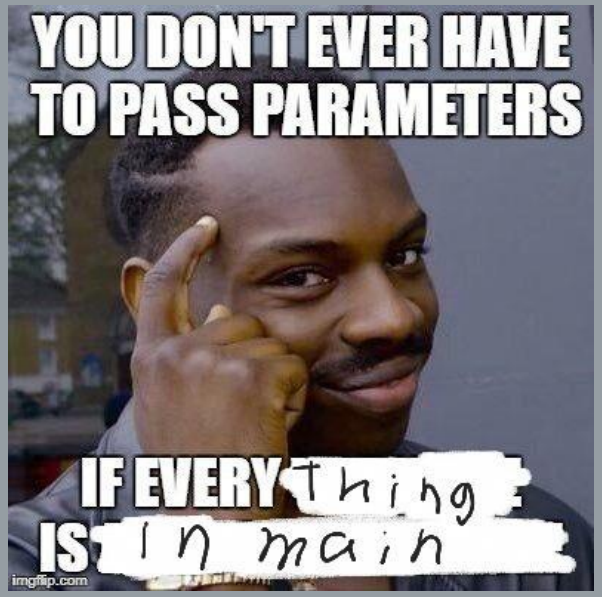






Parameter At Program Start

How can we use this?



Parameter At Program Start
Intro

Definition: Start parameters are inputs that are passed to a programme via the command line when it is started.

Benefit: They make it possible to flexibly change the behaviour of a program without having to adapt the code.

Examples: Passing file names, activating certain modes, configuring values.

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Number of arguments: %d\n", argc);
    for (int i = 0; i < argc; i++) {
        printf("Argument %d: %s\n", i, argv[i]);
    }
    return 0;
}
```

Parameter At Program Start

Intro

Information to a program can be passed at program start using arguments in the command line.

For passing the argument to the program there is a defined interface. The operating uses two formal parameters for it:

`argc`	argc (Argument Count)
`argv`	argv (Argument Vector)

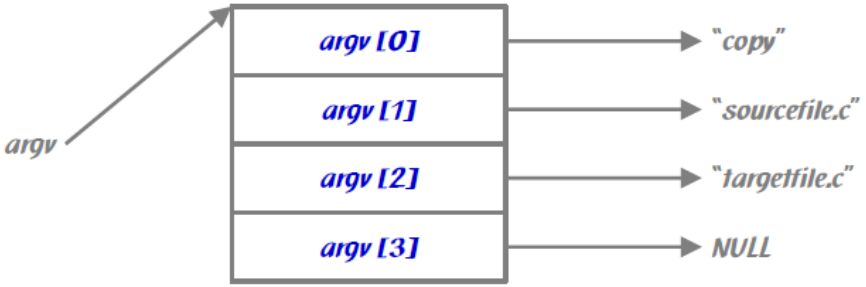
The counter of the parameter is at least `1`; the name of the program itself is the first argument.

Example:

Argument Vector

Text in command line:

```
copy sourcefile.c targetfile.c
```



Hint: All parameters are passed to the program as strings (character arrays). Number need to be transformed first.

Parameter At Program Start
Intro

Example:

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char * argv[])
{
    if (argc == 3) {
        printf ("The result is: \n");
        printf ("%s + %s = %d\n", argv[1], argv[2], atoi(argv[1]) + atoi(argv[2]));
    }
    else
        printf ("The correct program call is: parameter value1 value2\n");
    return (0);
}
```

Program Call:

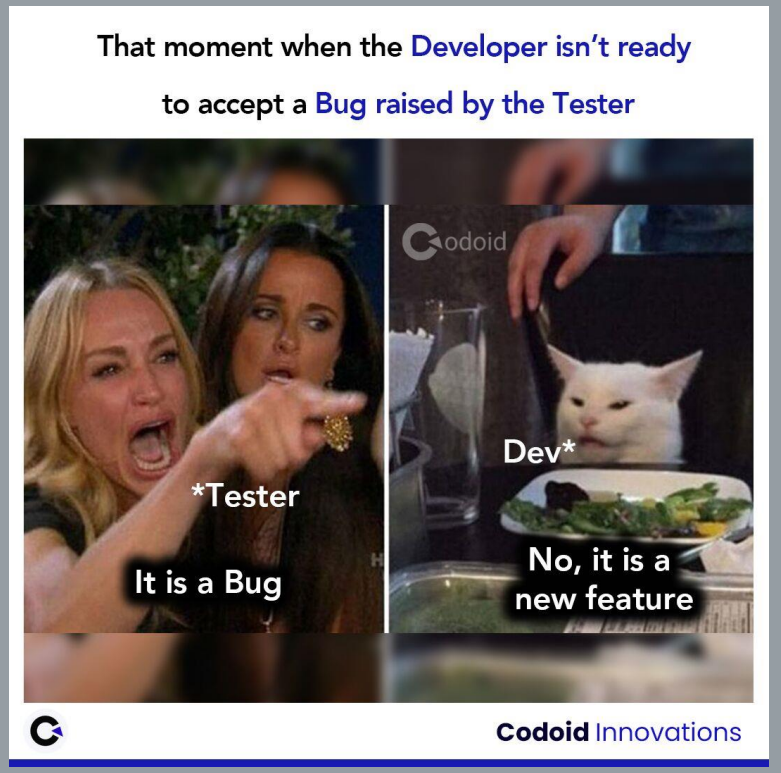
```
program 123 234
```

Screen Output:

```
The result is:
123 + 234 = 357
```

Testing

Why do we need Testing?



Testing
Function Prototypes <ctype.h>

The ctype.h library offers a range of functions for testing and working with characters.

```
int isalnum (char c);
```

The `isalnum` function tests a given integer value. It returns a non-zero value if the integer satisfies the test condition (here: alphanumeric), or a zero value if it does not. `c` must be presented as an unsigned character. `EOF` is a valid input value.

`int isalnum (char c);`	upper- lowercase letter or dec. digit
`int isalpha (char c);`	alphabetic character
`int iscntrl (char c);`	control character
`int isdigit (char c);`	decimal digit
`int isgraph (char c);`	printable character excluding space
`int islower (char c);`	lowercase letter
`int isprint (char c);`	printable character including space
`int ispunct (char c);`	any non-alpha printable character
`int isspace (char c);`	whitespace character
`int isupper (char c);`	uppercase letter
`int isxdigit (char c);`	hexadecimal digit

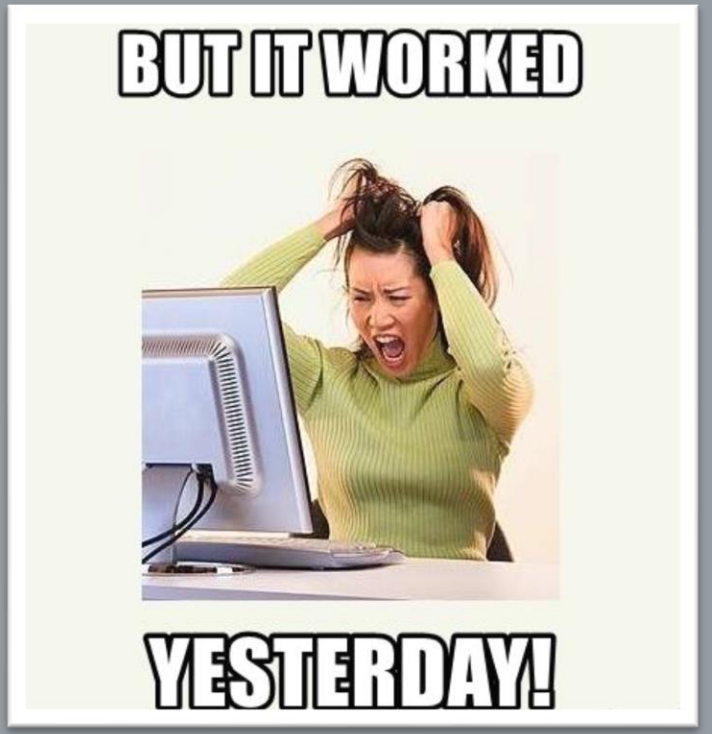
Testing
Function Prototypes <assert.h>

As you already know

- Assertion Error for Testing Using while developing a SW (Debugging Mode)
- useful during software development and testing to ensure that certain assumptions and conditions are met in the code

```c
#include <stdio.h>
#include <assert.h>

int main()
{
    int a = 1;
    int b = 2;

    assert(a == b);  // if a == b, nothing happens
                     // if a != b, program stops and prints error message

    return 0;
}
```

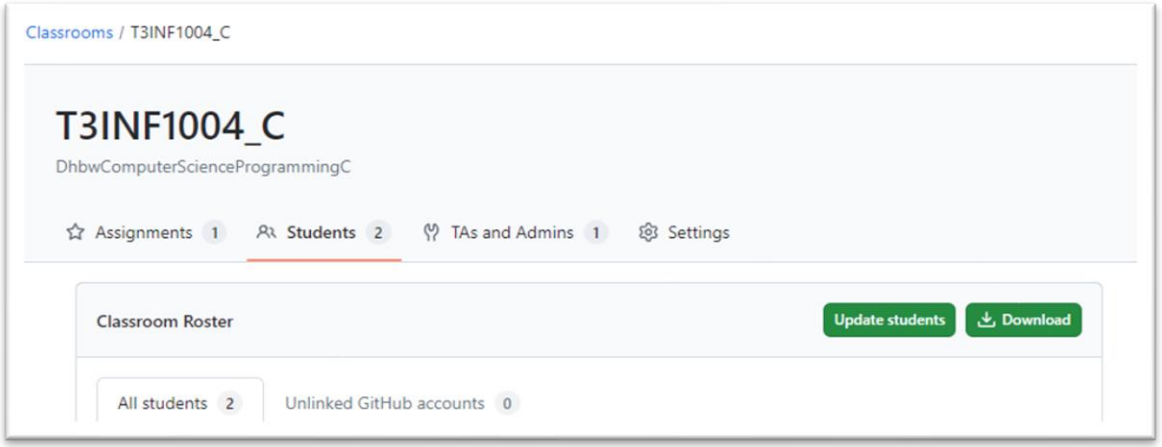
BACK TO GIT AGAIN



Live demo

Your Classroom for C coding Assignments

- Let's come together in the Classroom
- **02-Assignment** will be available for you
- Get the Repository
- **02-Assignment** Q&A



https://classroom.github.com/classrooms/182848101-t3inf1004_c/roster