Tutorial 06 - 14.12./17.12.2020

Group 02/11 - Moritz Makowski - moritz.makowski@tum.de

Multiple Files, `static`

Today's Agenda

- Using Multiple Files
  - Header Files
  - Shared Functions
  - Shared Variables/Constants
  - Shared Structs, Unions, Enums, etc.
- The `static` -Keyword
  - Static Global Variables
  - Static Local Variables

Why to Use Multiple Files?

- Separation of concerns
- Reduce complexity in large projects
- Make code available to other applications (e.g. libraries)

What is a library?

A library is a ...

> ... **set of functions** within a specific context, used by other programs, with the purpose of not having to **implement certain functionality** multiple times.

Examples: `math.h`, `stdio.h`, `stdlib.h`

Header Files - #1

Why are all these library names ending with `.h` and not `.c`?

That's because they are using so called "header"-files to manage what functions/constants are available from the outside.

Header Files - #2

Every `.c` -file that implements functionality which is used in some other `.c` -file should have a header-file.

It's best to have the same name for these two files. E.g.: `print_matrices.c` and `print_matrices.h`. Normally you have each `.h` and `.c` file in the same directory.

The basic structure of a header file:

```
#ifndef FILENAME_H
#define FILENAME_H

/* Your declarations go here */

#endif
```

Why to Use header files?

- Clear distinction between public and private
- To define a binding "contract" between two c-files (also called "interface")
- Prevent circular imports

Shared Functions - #1

`print_matrices.h`:

```
#ifndef PRINT_MATRICES_H
#define PRINT_MATRICES_H

// Only this method is meant to be shared
void print_int_matrix(int rows, int columns, int matrix[rows][columns]);

#endif
```

Shared Functions - #2

`print_matrices.c`:

```c
#include <stdio.h>

#include "print_matrices.h";

/**...*/
int longest_number_in_matrix(int rows, int columns, int matrix[rows][columns])
{/*...*/}

/**...*/
void print_int_matrix(int rows, int columns, int matrix[rows][columns])
{/*...*/}
```

Shared Functions - #3

`main.c`:

```c
#include "print_matrices.h"

int main() {
    int my_matrix[5][5] = {
        {12, -23, 40, 45},
        {33, 4, 0, 45},
        {12}
    };

    print_int_matrix(5, 5, my_matrix);

    return 0;
}
```

Shared Functions - Compiling

Compile these with:

```
gcc -Wall -Werror -std=c99 print_matrices.c main.c -o program.out
```

Notice that only the `.c` files are listed here!

Shared Functions - Example

Have a look at the directory `tutorial-06/example_6_1_shared_functions`.

The file `README.md` just contains the compilation string again.

Shared Variables/Constants - #1

`my_vars.h`:

```
#ifndef FILENAME_H
#define FILENAME_H

#define PI_APPROX 3.141592
// PI_APPROX is now accessible as a float in-
// side every file that includes this header

extern int global_var;

void increment_local_var(int amount);
void increment_global_var(int amount);

int get_local_var();

#endif
```

Shared Variables/Constants - #2

`my_vars.c`:

```c
#include "my_vars.h"

int local_var = 3;
int global_var = 42;


void increment_local_var(int amount) {
    local_var += amount;
}

void increment_global_var(int amount) {
    global_var += amount;
}

int get_local_var() {
    return local_var;
}
```

Shared Variables/Constants - #3

`main.c` :

```c
#include <stdio.h>
#include "my_vars.h"

int main() {

    printf("local_var: %d\n", get_local_var());
    printf("global_var: %d\n\n", global_var);

    increment_local_var(5);
    increment_global_var(5);

    printf("local_var: %d\n", get_local_var());
    printf("global_var: %d\n", global_var);

    // you cannot directly access "local_var" from this file
    return 0;
}
```

Shared Variables/Constants - #4

You can define **constants** directly inside the header-file.

These will be accessible in side every file that includes this header.

```
#define PI_APPROX 3.141592
```

Shared Variables/Constants - Compiling

Compile these with:

```
gcc -Wall -Werror -std=c99 my_vars.c main.c -o program.out
```

Notice that only the `.c` files are listed here!

Shared Variables/Constants - Example

Have a look at the directory `tutorial-06/example_6_1_shared_variables`.

The file `README.md` just contains the compilation string again.

Shared structs, unions, enums, etc. - #1

If you have structs/unions/... that "should only be"/"only have to be" accessible from within your `.c`-file then you can declare them inside your `.c`-file as well.

But when you declare them inside your header-file directly, they can be used in every file that includes the respective header-file.

Shared structs, unions, enums, etc. - #2

We will learn about stacks and pointers (`*` notation) shortly - `my_stack.h` :

```
#ifndef FILENAME_H
#define FILENAME_H

struct My_Stack {
  int *top;
  int *bottom;
  int *max_depth;
}

#endif
```

The `static` -Keyword

You can further limit the scope of your variables/functions by using the `static` -keyword.

It behaves differently for local and for global variables.

```
// regular variable
int var_a = 0;

// static variable
static int var_b = 4;
```

Static Global Variables

Static global variables or a function are only visible in the *current compilation unit* (for simplicity that is almost equal to the file scope).

Benefits:

- Limited access
- Compiler optimization: The compiler will produce more performant machine code

Static Local Variables

Static when used with local variables produces another variable that will - inside this scope - be used instead of the previously defined variable.

You can also say: Declaring a local static variable creates a new variable which "shadows" the global variable - with the same name - but has the same lifetime as the variable it is shadowing.

Static Local Variables - #2

I know that this sounds rather confusing!

Have a look at `example_6_4_static_local.c` and `example_6_5_static_global_*.c`!

Brain teaser: Have a look at `example_6_3_static_lifetime_.c` and try to figure out, what is going on.

You can comment out the `printf` statements for each variable individually to get a grasp of the procedure.

If you don't get it, it is totally ok! You can do almost everything you want without using static variables. "Almost" refers to some things not covered in this lecture.

See You Next Week!

All **code examples** and **exercise solutions** on **GitLab** (solutions right after my tutorial):

https://gitlab.lrz.de/dostuffthatmatters/IN8011-WS20

Me: *Spends two hours explaining how my code works*.
The person I'm explaining to: