1 Edit-Compile-Link-Execute

In the last exercise we used a single .c file and compiled it manually using *gcc*. However, in industry projects the code size can increase dramatically. Huge .c files are difficult to edit between multiple team members. Additionally, very large C projects take multiple hours to compile from scratch.

There exist multiple tools to simplify and optimise the build process. Since we are using *gcc*, we will use the *make* build system. *make* is based on build information stored in so-called *Makefile*s. Our main goal is to split the code into multiple .c, which can be edited independently. Furthermore, *make* should only recompile files when necessary, that is when they were edited since the last build.

This is called the "Edit-Compile-Link-Execute" process. Single .c files can be edited independently. Once the build process is started, these files are each compiled into a matching .o file. Once all files are compiled, these .o files are *linked* together to an executable program or a program library. The result can then be executed.

Open the file *Makefile*. This file contains all information necessary to build the project. *Makefile*s define build goals, which can be run using the command *make*. The default build goal is called *all*. *make* supports many helpful macros which simplify the build definition. *$(wildcard *.c)* defines a list of all .c files in the current directory. *$(addprefix build/,$(SRC:.c=.o))* adds a prefix to each of these list elements, which indicates that we want our immediate .o files stored in a directory called *build*.

In order to compile the project, run the following command on the console in the ex-02 directory:

```
> make all
```

Task 1 Which goals were executed? In which order?

Task 2 What is the purpose of the "clean" goal?

2 Pointer arithmetic

Pointers in C are references to variables and functions. They can be stored in pointer variables, allowing to access different memory locations depending on the current program state. Pointers have countless applications in C programs. Large input data can be provided by a single pointer value, avoiding the need to copy it.

Pointers can be type in the same manner like regular variables. The only additional syntax element necessary is the * operator. The following declares a pointer to an integer variable:

```
int *p;
```

Every variable can be addressed using the address-of operator &. This illustrated in the next listing:

```
int i = 999;
int *p = &i;
```

Pointers can be manipulated arithmetically. Their values can be incremented, decremented, added to or subtracted from. In order to access the value of the variable the pointer refers to, the value-of operator * can be used. The following example shows a pointer to an array element and increases the pointer to access the next element.

```
int i[] = { 0, 1, 2 };
int *p = i;
assert(*p == 0);
++p;
assert(*p == 1);
```

Arrays can be converted to a pointer of the same target type. The pointer then refers to the first element in the array.

Task 1 What is the pointer arithmetic equivalent to the array index expression *data[i]*?

Task 2 Implement a circular buffer

Implement a circular buffer in *read-write.c*. If your implementation is correct, the test cases in function *run_read_write_exercise()* should succeed.

Task 3 Sign Changes

Open *read-write.c* and implement the required function, which takes a pointer to an integer, and changes the integer's sign. Test should all pass if the function is implemented correctly.

3 Stack and Heap

The C programming language provides two fundamental idioms of memory provision. The first one is the stack. The stack provides automatic memory. We have been using the stack during all of the previous exercises. The stack, as its name suggests, grows linearly with every function called by the current thread. Apart from some additional metadata, the stack adds a frame for every function called. The frame is large enough to hold the local variables declared in the called function.

Once the function returns control to its calling function, its stack frame is automatically destructed. This means that the lifetime of variables on the stack is strictly limited by a scope and cannot be extended beyond that. However, this is required in some cases.

As an example, in certain algorithms we want to be able to allocate an amount of memory in one function and keep the memory valid beyond the function's execution. We then initialise and manipulate the data in another function. Once our algorithm is complete, we manually delete the memory in a final clean-up function. This scenario is where heap memory comes into play in C. The following listing shows an example of heap memory allocation in C:

```
int *reserve_memory() {
  return (int *) malloc(100 * sizeof(int));
}

void initialise_memory(int *data) {
  int i;
  for(i = 0; i < 100; ++i) {
    data[i] = 0;
  }
}

void manipulate_data(int *data) {
  data[0] = 1;
  data[1] = 1;
  int i;
  for(i = 2; i < 100; ++i) {
    data[i] = data[i - 2] + data[i - 1];
  }
}

void release_memory(int *data) {
  free(data);
}
```

The standard C method *malloc(...)* allows us to allocate memory of a requested size on the heap storage. The function returns a pointer to the allocated memory, which needs to be cast to its intended type. This memory is not limited in scope and will remain valid until it is explicitly released using the funtion *free(...)*.

Task 1 Transfer the given code to the file *malloc-free.c*

Task 2 Use these methods to print the Fibonacci series.

4 Illegal Memory

Pointer arithmetic represents a substantial tool in C programming and is indispensable in order to write efficient C programs. However, they are also the cause of serious program errors. Pointers can refer to unexpected memory locations, leading to unintended data manipulations. They may even point to completely inaccessible memory locations, leading to program crashes.

```
int *p = 0;
printf("%d\n", *p); // Undefined behaviour!
int *p = (int *) 0x12345678;
```

```
printf("%d\n", *p); // Undefined behaviour!
```

A pointer value of *0* is reserved to indicate that the pointer does not refer to any memory. Dereferencing a null pointer leads to undefined behaviour. Undefined behaviour refers to situations which are not covered by the C standard. These lead to unexpected program behaviour and can even cause the program to crash.

Task 1 Fix errors in code.

We deliberately introduced multiple pointer errors in the file *illegal-memory.c*. Try to run the function *run_illegal_memory_exercise()* and inspect the source code. Describe and eliminate all errors present.

Task 2 What is the result of freeing a *null* pointer?

Explain what best practice you can take away from this.

5 Function pointers

Function pointers are references to C functions. This concept allows the assignment of target functions to variables. This is used to adapt the function call dynamically in a program. They are declared along the following pattern:

```
return_value (*pointer_variable_name)(argument_type1, argument_type2, ...)

// Example:
void (*fp)(int, double);
```

function-pointers.c shows an example of function pointer usage.

Task 1 Explain the code in *run_function_pointers_exercise()*.

Task 2 Replace the type declarations in *execute_output(...)* by actual function pointer types.

6 Callbacks

A common example for function pointer usage are callback registrations. It allows users to dynamically configure which functions should be called if a certain event occurs. Use the stub in *callbacks.c* to create a callback registry.

Task 1 What is the expected output of *run_callbacks_exercise()*?

Task 2 Implement the callback registry.

Use the function stubs and variables as indicated in the code comments. Create an array for the registered functions pointers. Mark slots in the array which are not currently used with null pointers.

7 Fetch-decode-execute cycle

We learned about the fetch-decode-execute cycle in processor pipelines. Generally the intent of higher-level programming languages is to hide away hardware details and allow programmers to focus on the actual behaviour they want to implement. However, understanding underlying concepts can be important in situations where they affect a program's behaviour or performance.

Task 1 Run scenario 1 and 2 pipeline.c

Which one is faster? Are the results surprising if you inspect the source code?

Task 2 A processor pipeline optimisation is causing this discrepancy. Explain which one it is and how it affects this experiment.

8 Data structures

In exercise 1 we identified a plane coordinate using two variables x and y. Larger software systems sometimes require the definition of data structures which are logically composed of multiple components. An example is a data structure which describes personal information. Such a structure needs to store first names, last names, social security numbers, etc. C provides *struct* and *union* data types for this form of composed information.

Inspect the code in *data-structures.c*.

Task 1 Draw the hierarchical data structure suggested by the initialiser in *create_point()*.

Task 2 Implement the data structures accordingly.

Task 3 Implement the missing operations in order to make the test cases pass.