# Laboratory exercise 5.

*Flow control*

## Purpose of the exercise

This exercise will help you do the following:

1. Practice developing programs that include complex flow control statements.
2. Develop familiarity with expressions and operators.
3. Develop familiarity with more advanced use cases of pre-processor directives in C programs.

## Overview

### Statements and expressions

In C programs you control the flow of operations by using different kinds of statements. Statements are fragments of a program that are executed in sequence. They fall into the the categories shown below:

- Expression statements,
- Selection statements,
- Iteration statements,
- Jump statements,
- Block statements (sometimes called compound statements),
- Empty statements (sometimes called *null* statements and considered a special case of expression statements without an expression).

Statements let functions perform calculations, branch the code, jump ahead or backwards. Familiarity with the syntax (*how to write them*) and semantics (*how to understand them*) is a crucial skill for every programmer.

Many of these statement incorporate expressions, sequences of operators and their operands. They are the main building blocks of expression statements, but also represent conditions in selection and iteration statements, or returned values of some jump statements. It is nearly impossible to write a non-empty program that does not use expressions. Operators used in the expressions generally belong to the following categories:

- Assignment (simple `=` and compound, i.e. `+=`),
- Increment/decrement (in prefix and postfix versions),
- Arithmetic (including bitwise),
- Logical,
- Comparison,
- Member access (we will cover them later in the course),
- Other (including a function call, `,`, `sizeof`, ternary operator `?:` and type casting).

This time you will combine complex statements and complex expressions to develop an interactive calculator, where users can evaluate expressions input in a format `OPERAND1 SYMBOL OPERAND2`, for example `1 + 2`.

## Conversions and casting

When an expression is used in an assignment, initialization, function call or `return` jump statement and a resulting data type is different than expected in that context, a result of the expression must be converted to an expected data type. When a binary operator is used with operands of different types, a conversion is expected to bring both operands to their common data type.

Conversion can be *explicit*, indicated in code by a programmer, or *implicit* - handled by a compiler. Depending on data types of operands included and a context, a compiler may perform *usual arithmetic conversions*, integer promotions, or other types of implicit conversions. On the other hand, a programmer can use a cast operator to perform an explicit conversion, also known as *casting*, to a type of their choice. For example, to explicitly convert an `int` literal to a `short int` the cast operator can be used within an initialization statement like so:

```
short int x = (short int)10;
```

Conversions can lead to loss of data or sign. Suppose that the following code is compiled for Microsoft Windows 10 on a 64 bit CPU architecture:

```
unsigned int i = 0xFFFFFFFF;    // 32 bits all set to 1
short int x = i;                // Data type with 16 bits only
```

This implicit conversion will lead to data truncation: most significant 16 bits will be lost; as a result the value will also become negative. To receive compilation warnings that protect you from unintended risky implicit conversions compilers offer appropriate flags; in *gcc* you can add an option `-Wconversion`. A generated compilation warning may look like this:

```
conversion from ‘unsigned int’ to ‘short int’ may change value
```

Alternatively, if this conversion is indeed intended by a programmer, an explicit conversion can be used:

```
unsigned int i = 0xFFFFFFFF;
short int x = (short int)i;     // Explicit conversion
```

## Preprocessor

A preprocessor is a program (or a functionality of a compiler driver itself) executed before a compilation phase. Its task is to resolve preprocessing instructions, *directives*, embedded inside the code and to produce a single translation unit ready for compilation.

Although a preprocessor is very limited in what it can do, for the C programming language it solves a few very important challenges:

- Includes external files containing fragments of the translation unit.
- Conditionally returns or omits parts of code (based on own simple expression language).
- Performs substitution of text identifiers, *macros*, with their values (thus offering compile-time constants).

Earlier in this course you have learnt how to implement the first of these capabilities and use a directive `#include` to include standard headers and header files. This exercise shows you two latter behaviors.

Firstly, take a look at *operation.h* in the **Step 2**. Most of the lines there have a format:

```
#define IDENTIFIER REPLACEMENT
```

An example is shown below:

```
#define ADDITION (operation)1
```

This line defines a macro with an identifier `ADDITION` and instructs the preprocessor to replace all subsequent instances of this identifier with a replacement text `(operation)1`; this replacement value represents a cast expression converting an integer literal `1` into a data type `operation`. Such casting may be useful as macros are not strongly-typed: they do not have a data type as they are just a simple text replacement instructions.

The replacement happens before compilation, and a compiler will use this expression as a compile-time constant; a result of casting of a literal is seen as equivalent to a literal of a target data type. Take a note that a preprocessor macro is not a statement. `#define` is not terminated by a semicolon, unless this character should be a part of a replacement text.

While `#define` is used for introducing new macros to the code, to remove a macro it has to be undefined with a directive `#undef`, i.e. `#undef IDENTIFIER`.

Secondly, notice that at the beginning of the file and at the end of the file there are two different directives used together with `#define`:

```
#ifndef _OPERATION_H_
#define _OPERATION_H_

//...

#endif // _OPERATION_H_

```

A directive `#ifndef` (*if not defined*) and its matching `#endif` delimit a fragment of code that will be included in a produced translation unit only of a given identifier has not yet been defined.

This code represents a very common pattern called the **header guards** that lets C programmers avoid including the fragment twice and thus introducing duplicated definitions. An identifier's name is derived from the name of the file, here `_OPERATION_H_`. If the current header file has not yet been included, the identifier has not been defined, and a preprocessor will include the fragment between `#ifndef` and `#endif`; by doing so it will also define the identifier with `#define`. Next time the file is being included, the directive `#ifndef` will find an existing identifier and omit the code preceding `#endif`. The single-line comment is not necessary, but as the `#endif` is typically placed at the end of the file, it informs other programmers about the macro related to this condition directive.

In this exercise, the code inside *main.c* includes *calc.h* and *operation.h*; *calc.h* also does include *operation.h* as it needs a declaration of a data type `operation`. This organization of the code could lead to redefinitions of included macros, which could be at least unnecessary, and in a more complex scenario make the code ill-formed or uncompilable.

## Tasks

In this exercise you will complete a program that consists of two separate translation units.

The first unit is implemented in a file *main.c* and represents the main driver of the program with the definition of the function `main()`. A fragment of this program has been provided to you in this exercise.

The second unit that will reside in *calc.c* should contain a definition of the function `calculate()`. This file will be created by you.

Therefore, the program will have the following structure:

- *main.c* - the entry point of the program will handle user input, validation, and selection of the operation to be performed; a part of your task is to complete this program.
- *calc.c* - a source file where the function `calculate()` is defined.
- *operation.h* - a header file that defines `operation` data type that in your program will represent a type of an arithmetic operation to be performed. Even though this type is just an alias to a small integer data type `unsigned char`, giving it a name *abstracts* away that implementation detail, and let's you focus on the role of this type rather than its representation. The file also contains preprocessor definitions of compile-time constants for all required operations. This file has been provided to you; do not change it, but get familiar with its contents in detail.
- *calc.h* - a header file that provides a declaration (in fact, a full prototype) of the function `calculate()` that performs all operations. This file has been provided to you; do not change it, but get familiar with its contents in detail.

In this exercise you will complete the implementation of an interactive calculator by adding the missing code into *main.c* and creating *calc.c* from the start.

## Step 1. Prepare your environment

Open your WSL Linux environment, prepare an empty *sandbox* directory where the development will take place, create a new file *operation.h* and open it for editing:

```
code operation.h
```

This command should open Microsoft Visual Studio Code for editing your code.

## Step 2. Edit *operation.h*

Replicate the following code into the file *operation.h*:

```c
#ifndef _OPERATION_H_
#define _OPERATION_H_

typedef unsigned char operation;

#define UNKNOWN (operation)0
#define ADDITION (operation)1
#define SUBTRACTION (operation)2
#define MULTIPLICATION (operation)3
#define DIVISION (operation)4
#define DIVISION_INTEGERS (operation)5
#define MODULUS (operation)6

#endif // _OPERATION_H_

```

Save the changes. You will not need to edit or submit this file, but it will be used during compilation. Make sure you understand every statement in this file. Take note of the *header guards* included in the file.

## Step 3. Edit *calc.h*

In the same way as *operation.h*, create *calc.h* and replicate into it the following code:

```c
#ifndef _CALC_H_
#define _CALC_H_

#include "operation.h"

void calculate(float x, float y, operation op);

#endif // _CALC_H_

```

Save the changes. You will not need to edit or submit this file, but it will be used during compilation. Make sure you understand every statement in this file. Take note of the *header guards* included in the file.

## Step 4. Edit *main.c*

In the same way as *operation.h* and *calc.h*, create *main.c* and replicate into it the following code:

```c
#include <stdio.h>
#include "calc.h"
#include "operation.h"

int main(void)
{
    printf("This program evaluates mathematical expressions.\n");

    /* TODO */

    printf("Closing the program...\n");
    return 0;
}

```

The `/* TODO */` part will have to be replaced with the code for handling interactions with the user, but to test the program in its most simple form start by replacing it with the call to the function `calculate()` with known values and the operation, for example:

```c
calculate(2.0f, 3.0f, ADDITION);
```

Take note that the last parameter is a preprocessor macro called `ADDITION` defined in *operation.h* that will resolve into `(operation)1`.

## Step 5. Add file-level documentation

In the **Laboratory exercise 3.** you have learnt that every source code file you submit for grading must start with a file-level documentation header. Open the specification of that exercise and use the template of a header provided there in *main.c*. Replace `@todo` and the rest of each line with your information. When you edit the file later, remember to check if the information is up-to-date.

The file-level documentation should precede any content of a source file (including the header guards appropriate for all header files).

## Step 6. Add function-level documentation

Since all functions in this exercise are well defined, you are ready to write their function-level documentation.

In the **Laboratory exercise 3.** you have learnt that every function in every source code file you submit for grading must be preceded by a function-level documentation header that explains its purpose, inputs, outputs, side effects and lists any special considerations. Open the specification of that exercise and review the example of a function header documentation provided there; add relevant function-level documentation for all functions in *main.c*.

## Step 7. Compile *main.c*

You are now able to compile (but not link) an object file from *main.c*:

```
gcc -Wall -Werror -Wextra -Wconversion -Wstrict-prototypes -pedantic-errors -
std=c11 -c -o main.o main.c
```

If the compilation resulted in any errors, resolve them and try again.

If there are no issues, it is time to implement *calc.c*. You will need to return to the file *main.c* and edit it later.

## Step 8. Edit *calc.c*

Similarly to *main.c*, create *calc.c*. This source file will have to perform printouts and define the function `calculate()` matching the declaration from *calc.h*. This means it has to include the following headers:

- *<stdio.h>*
- "calc.h"

With your knowledge of C, you should already know how to add them.

Then copy the prototype of the function `calculate()` and provide its stub implementation that handles nothing but addition (`ADDITION`). This function does not return anything, but it is expected to print out a single tab character, followed by a floating-point result of adding both parameters, followed by a new line character.

## Step 9. Add file-level documentation

In the **Laboratory exercise 3.** you have learnt that every source code file you submit for grading must start with a file-level documentation header. Open the specification of that exercise and use the template of a header provided there in *calc.c*. Replace `@todo` and the rest of each line with your information. When you edit the file later, remember to check if the information is up-to-date.

## Step 10. Add function-level documentation

Since all functions in this exercise are well defined, you are ready to write their function-level documentation.

In the **Laboratory exercise 3.** you have learnt that every function in every source code file you submit for grading must be preceded by a function-level documentation header that explains its purpose, inputs, outputs, side effects and lists any special considerations. Open the specification of that exercise and review the example of a function header documentation provided there; add relevant function-level documentation for all functions in *main.c*.

## Step 11. Compile *calc.c* and link with *main.c*

You are now able to compile an object file from *calc.c*:

```
gcc -Wall -Werror -Wextra -Wconversion -Wstrict-prototypes -pedantic-errors -
std=c11 -c -o calc.o calc.c
```

If the compilation resulted in any errors, resolve them and try again.

Because both translation units are compileable, you can also compile and link them together to create an executable:

```
gcc -Wall -Werror -Wextra -Wconversion -Wstrict-prototypes -pedantic-errors -
std=c11 -o main main.c calc.c
```

## Step 12. Test partial implementation

The program right now should just start, print the result of the addition and terminate. If you achieved this behavior, it should be straightforward to add implementation for all missing operations.

## Step 13. Implement all operations

Edit *calc.c*. You will need to add a selection statement to perform different operations, based on the third parameter passed into a function; for this purpose use of the `switch` statement. If you need a refresher about its syntax, review the reference documentation: https://en.cppreference.com/w/c/language/switch

The following behavior is expected:

- `ADDITION` - `float` addition,
- `SUBTRACTION` - `float` subtraction,
- `MULTIPLICATION` - `float` multiplication,
- `DIVISION` - `float` division; in case of division by 0 print `Division by 0!`,
- `DIVISION_INTEGERS` - `int` division; both operands must be cast to `int`; in case of division by 0 print `Division by 0!`; cast the result back to `float` before printing,
- `MODULUS` - remainder of `int` division; both operands must be cast to `int`; in case of division by 0 print `Division by 0!`; cast the result back to `float` before printing,
- Any other operation - print `Unknown operation selected!`.

Each time you add a support for an operation adjust *main.c* to test it. Repeat this process until all operations are implemented correctly.

## Step 14. Add user interactivity

The program will have to continuously prompt the user for an expression. To achieve it, inside the function `main()` use an infinite loop. All iteration statement types can be written to never terminate:

```c
// An infinite `for` loop
for (;;)
{
    // DO SOMETHING
}
```

```c
// An infinite `while` loop
while (1)
{
    // DO SOMETHING
}
```

```c
// An infinite `do/while` loop
do
{
    // DO SOMETHING
}
while (1);
```

Within the loop display a message asking the user the enter an expression, and with `scanf()` capture the input consisting of whitespace-separated `float`, `char`, `float` inputs, where the `char` represents the symbol, while both `float` values represent operands. Do check the file *expected-output.txt* for available symbols and the exact text of all text messages. After capturing the user input perform checks:

- If the user entered all 3 values, use the `char` representing the symbol to set the variable that indicates the operation type, then call the `calculate()` function.
- Otherwise, if at least 1 variable was captured, print `Invalid number of arguments!`, clear the input buffer by reading remaining characters to the end-of-line (`\n`) or the end-of-file (`EOF`), and prompt the user again for input. To clear the buffer you can use the code:

  ```c
  int c;
  while ((c = getchar()) != '\n' && c != EOF);
  ```

- Otherwise, jump out of the loop using `break` and terminate the program.

## Step 15. Complete the program and clean-up the code

Now you have all the building blocks - you need to put them together to create a program that behaves in a required way.

### Step 16. Test

Compile and link your program using the full suite of required *gcc* options:

```
gcc -Wall -Werror -Wextra -Wconversion -Wstrict-prototypes -pedantic-errors -std=c11 -o main main.c calc.c
```

Then run the executable:

```
./main
```

Test few examples entered by hand and check if the output is properly formatted and matches your expectations.

In this exercise you have been provided with a file *input.txt*. Execute the program and generate *actual-output.txt*:

```
./main < input.txt > actual-output.txt
```

You are also given an output file *expected-output.txt* that contain the correct result. Your actual output must exactly match the contents of the expected output. Use the *diff* command in the Linux bash shell to compare the output files:

```
diff --strip-trailing-cr -y --suppress-common-lines actual-output.txt expected-output.txt
```

If *diff* completed the comparison without generating any output, then the contents of the two files are an exact match.

## Submission

Once your implementation of *main.c* and *calc.c* is complete, again ensure that the program works and that it contains updated file-level and function-level documentation comments.

Then upload the files to the laboratory submission page in Moodle. There are 50 lines of expected output and 50 corresponding automated test cases. To get the maximum grade, make sure that all test cases match.

## If you're done early and are bored...

If you finish ahead of time, declare an array of 10 `float` elements in the function `main()`, update `calculate()` to accept this array and its size, and after every successful calculation update the array so that it contains 10 most recent calculation results. Try to display this array before prompting the user for the next expression to evaluate.

**Do not submit this code!**