Laboratory 2

1. General data types

Type	Data type size	Value range
`char`	1 byte	[-128, 127] or [0, 255]
`int`	2 or 4 bytes	[-32.768, 32.767] or [-2.147.483.648, 2.147.483.647]
`float`	4 bytes	[1,2E-38, 3,4E+38]
`double`	8 bytes	[2,3E-308, 1,7E+308]

- data types `char` and `int` can also be qualified with the keyword: `unsigned`
- data type `int` can also be: `short` or `long`

Type	Data type size	Value range
`unsigned char`	1 byte	[0, 255]
`unsigned int`	2 or 4 bytes	[0, 65.535] or [0, 4.294.967.295]
`short (int)`	2 bytes	[-32.768, 32.767]
`unsigned short`	2 bytes	[0, 65.535]
`long (int)`	4 bytes	[-2.147.483.648, 2.147.483.647]
`unsigned long`	4 bytes	[0, 4.294.967.295]

2. Keywords

`auto`	`break`	`case`	`char`	`const`	`continue`
`default`	`do`	`double`	`else`	`enum`	`extern`
`float`	`for`	`goto`	`if`	`int`	`long`
`register`	`return`	`short`	`signed`	`sizeof`	`static`
`struct`	`switch`	`typedef`	`unsigned`	`union`	`void`
`volatile`	`while`				

3. Constants

- Defined in two ways:
  - Using `#define`: (Technically not constants, but their value remains the same)

```
#define TEN 10
#define NEWLINE '\n'
```

  - Using `const`:

```
const int TEN = 10;
const char NEWLINE = '\n';
```

4. Variables

- defining variables:

```
data_type  var_name;
```

  - *data_type* can be anything shown in the section “General data types” and more
  - ***var_name*** can contain alphanumeric characters and the „ _” *(underscore)*
  - variable names must begin with a letter
  - variable names cannot be keywords (see the “Keywords” section)
  - C is case-sensitive

- examples:

```
int n;                       int n = 10;
char c;                      char c = 'a';
```

5. Operators

- Types:
  - *Arithmetic:* `+  -  *  /  %  ++  --`
  - *Relational:* `==  !=  >  <  >=  <=`
  - *Logical:* `&&  ||  !`
  - *Bitwise:* `&  |  ^  ~  <<  >>`
  - *Assignment:* `=  +=  -=  *=  /=  %=  <<=  >>=  &=  ^=  |=`
  - *Others:* `sizeof()  &  *  ?:`
- Priorities:

`https://en.cppreference.com/w/c/language/operator_precedence`

6. Derived data types

6.a. Arrays (vectors)

```
data_type array_name[size];
```

- examples:

```
int arr[5];
int arr[5] = {10, 20, 30, 40, 50}; // with initialisation
double values[] = {100.0, 2.0, 300.0, 40.0, 50.0}; // no given size, but
can be deduced from initialisation
```

6.b. Strings

```
char msg[] = "Hello";
char msg[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

6.c. Pointers

- *pointer* = *a variable that contains the address of another variable*
- examples:

```
int *p;    // pointer to an integer variable
char *c;   // pointer to a char variable
```

```
float *f;  // pointer to a float variable
double *d; // pointer to a double variable
```

- Obtaining the address of variable `x`: `&x`
- Obtaining the value (dereferencing) stored at the address specified by `x`: `*x` *(if x is a pointer type)*

6.d. Structures (Structs)

- Defining a struct:

```
typedef struct {
     int id;
     char author[50];
     char title[100];
} Book;
```

- Declaring a struct variable and using it:

```
int main(int argc, char** argv)
{
     Book book1;
     ...
     book1.id = 1000;
     strcpy(book1.author, "B.W. Kernighan, D.M. Ritchie");
     strcpy(book1.title, "The C Programming Language");
          ...
          return 0;
}
```

7. Functions

- Defining a function:

```
return_type function_name(data_type param1, data_type param2, ...);
```

where:

  - `return_type` can be any general data type/derived data type or `void`
  - **`function_name`** same restrictions as variable names
  - `data_type` **`param1`**, `data_type` **`param2`**, `...` are called formal parameters
  - the function may have 0 parameters, in which case there is nothing between the round brackets at definition

- examples:

```
void print_matrix(int** mat)

float average(int a, int b)

int** read_matrix_from_file(FILE* file)

int generate_random_int()
```

- `main()`:
  - The entry point -> program begins here
  - The prototype used for this subject must be:

```
int main(int argc, char** argv)
```

- returns an **int,** that indicates if the process execution was successful or not (0 means success, any other value is some sort of an error/warning)
- has **2 arguments:**
  - **argc ->** the number of command-line arguments (the command itself is counted as well)
  - **argv ->** array of strings, each element is one argument

(argv[argc] == NULL; argv[argc+1] -> index out of bounds)

8. I/O functions

```
int getchar(void)
int putchar(void)
char *gets(char *s)
int puts(const char *s)
int scanf(const char *format, ...)
int printf(const char *format, ...)
```

9. Interacting with files

9.a. Text files:

```
FILE *fopen(const char *filename, const char *mode)
int fgetc(FILE *fp)
int fgets(char *buf, int n, FILE *fp)
int fputc(int c, FILE *fp)
int fputs(const char *s, FILE *fp)
int fscanf(const char *format, ...)
int fprintf(const char *format, ...)
int fclose(FILE *fp)
```

9.b. Binary files:

```
size_t fread(void *buf, size_t bsize, size_t nbyte, FILE *fp)
size_t fwrite(const void *buf, size_t bsize, size_t nbyte, FILE *fp)
```

or:

```
int open(const char *path, int oflag, ... )
ssize_t read(int fd, void *buf, size_t nbyte)
ssize_t write(int fd, const void *buf, size_t nbyte)
int close(int fd)
```

10. Basic examples

```
// Print all arguments received from the command line
#include <stdio.h>
int main(int argc, char **argv) {
     int i;
     for(i = 0; i < argc; i++) {
          printf(“%s\n”, argv[i]);
     }
```

```
      return 0;
}
```

```
// Print a greeting message to a name given as an argument
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
      if(argc < 2) {
            printf(“Please provide at least one argument\n”);
            exit(1);
      }
      printf(“Hello, %s\n”, argv[1]);
      return 0;
}
```

!! C programs **must** be compiled using:

a. **gcc -Wall -g sourceCode.c -o programName**

- **-Wall** -> displays all compilation warnings and errors (all warnings/errors must be resolved)
- **-g** -> adds debugging information to the executable (useful if using gdb)
- **sourceCode.c** -> the file containing the source code that needs to be compiled - can be named whatever you want
- **-o programName** -> will store the compilation output (a program) in a file named “programName” - this can be named whatever you want

b. If compilation was successful, run with

```
./programName arg1 arg2 arg3 …
```

and

```
valgrind ./programName arg1 arg2 arg3 …
```

Valgrind will report any memory mismanagement (unallocated memory, unclosed files, using variables before allocation, etc.) - **All issues reported by valgrind must also be resolved.**

11. Potential warnings/errors

- syntax errors
- missing include files
- using an undefined variable
- redefinition of the same variable
- using an undefined function

- calling a function using an incorrect definition (wrong number of arguments, wrong order of arguments)
- using an uninitialised pointer variable
- not opening files before read/write
- opening files in the incorrect mode (eg: open for read, but trying to write)
- forgetting to deallocate pointers
- forgetting to close files