Prerequisites:

- Basic knowledge of using computer and text editor
- Knowledge how to create a program
- Knowledge how to create functions
- Knowledge of different types

Aims:

In this laboratory student will learn how to:

- Reading from console
- Logical operators
- Checking conditions
- Usage of `if`, `else`, `switch`, `enum`

1 Theoretical background

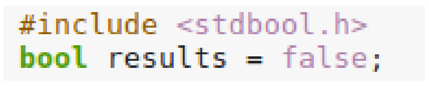
1.1 Logical Values

Like in mathematics in informatics there are only two logical values:

- **false** → has value 0 and corresponds to the inactive state,
- **true** → has value 1 and corresponds to the active state.

It should be noted, that these values corresponds exactly to two possible bits' states in binary system.

In the C language there is a special type to represent logical values. It is called **bool**. Variables of that type can be only `false` or `true`. To use such a type in program a special library `stdbool.h` has to be included in your program (see Listening 1.1).



```
#include <stdbool.h>
bool results = false;
```

Listing 1.1: Usage of bool-type's variable

1.2 Logical Expressions

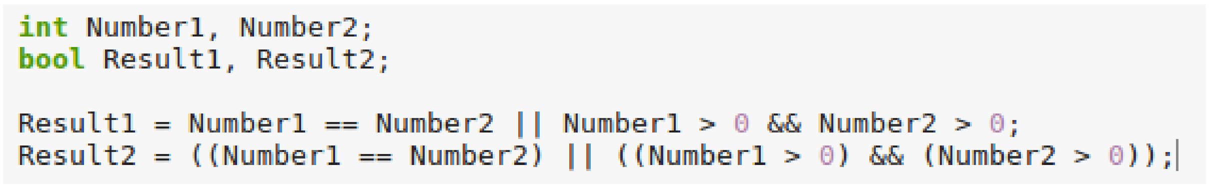
Logical expressions are in facet a algebraic expressions which result is `true` or `false` (1 or 0 in bits, respectively). Because logical operations in the C language differ from arithmetical operations, similarly as in mathematics, a special set of logical operators is given (see Tab. 1.1). Logical operators can be merged in larger chains or grouped using braces. Order of operations is similar as in mathematics:

1. expressions in braces,
2. negation,
3. comparison of logical values,
4. modulo sum,
5. logical product,
6. logical sum.

It is advised to group ingredients of larger logical expressions using braces, even if they are not needed. Take a look closer to example given in Listening 1.2. Both expressions are equal to each other, however, the second one is easier to read, follow, and understand.

Tab. 1.1: Logical operators in the C language

symbol	name	example	meaning
==	equality	`a == b`	Returns `true` if both compared values are equal to each other; in other case it returns `false`. Both variables have to be this same type.
`!=`	inequality	`a != b`	Returns `true` if both compared values are different; in other case it returns `false`. Both variables have to be this same type.
<	less than	`a < b`	Returns `true` if the first value is less that the second one; in other case it returns `false`. Both variables have to be this same type.
<=	less or equal	`a <= b`	Returns `true` if the first value is less that or equal to the second one; in other case it returns `false`. Both variables have to be this same type.
>	greater than	`a > b`	Returns `true` if the first value is greater that the second one; in other case it returns `false`. Both variables have to be this same type.
>=	greater or equal	`a >= b`	Returns `true` if the first value is greater that or equal to the second one; in other case it returns `false`. Both variables have to be this same type.
!	negation	`!a`	Returns logically opposite value to the given one: `true` if `a` is `false` or `false` if `a` is `true` The `a` has to by a logical value.
`||`	sum	`a || b`	Returns `true` if any of variables has value `true`; in other case it returns `false`. Both values have to be logical values.
&&	product	`a && b`	Returns `true` if both variables have value `true`; in other case it returns `false`. Both values have to be logical values.
^	modulo sum	`a ^ b`	Returns `true` if only one variable has value `true`; in other case (both variables are `true` or `false`) it returns `false`. Both values have to be logical values.



```c
int Number1, Number2;
bool Result1, Result2;

Result1 = Number1 == Number2 || Number1 > 0 && Number2 > 0;
Result2 = ((Number1 == Number2) || ((Number1 > 0) && (Number2 > 0));
```

Listing 1.2: Using braces to separate logical expressions

Be alerted that operators = and == are different! Replacing one of them to other one is not an error in terms of C language's syntax, nevertheless, it changes sens of the program!

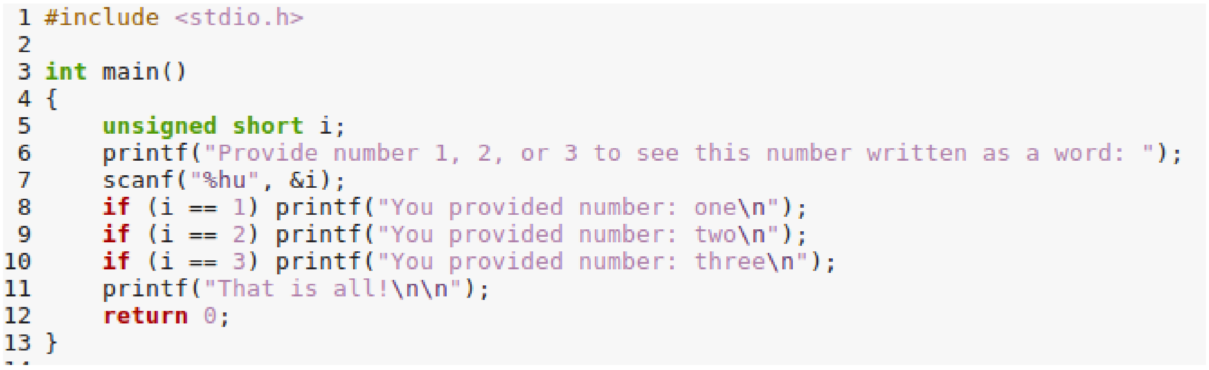
1.3 Conditional Statements

Conditional statement allows to choose one from few ways the program could be running. It depends if a statement given by a programmer and expressed by a logical expression is fulfilled or not. In the C language such a condition statement is called `if`

```
if (logical_expression) something_to_do;
```

If `logical_expression` has value `true` then `something_to_do` will be executed. Otherwise `something_to_do` will be omitted (it means that `logical_expression` has value `false`).

Take a look at a simple example given in Listening 1.3. As you can see this program is not an optimal one. All three `if` statements are always checked, even if one of them is true. What is more, if user provide different value than 1, 2, or 3, this program cannot handle this because such a possibilities is not handled.

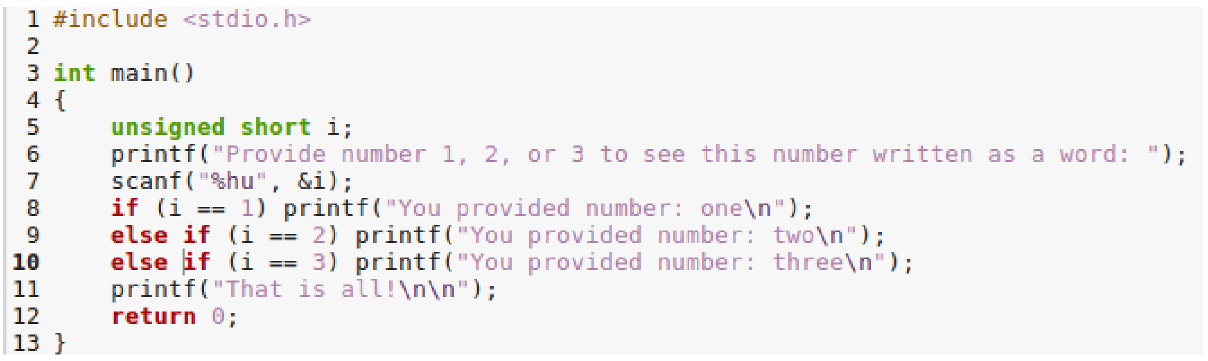


```c
#include <stdio.h>

int main()
{
    unsigned short i;
    printf("Provide number 1, 2, or 3 to see this number written as a word: ");
    scanf("%hu", &i);
    if (i == 1) printf("You provided number: one\n");
    if (i == 2) printf("You provided number: two\n");
    if (i == 3) printf("You provided number: three\n");
    printf("That is all!\n\n");
    return 0;
}
```

Listing 1.3: Example usage of a conditional statement using the if instruction

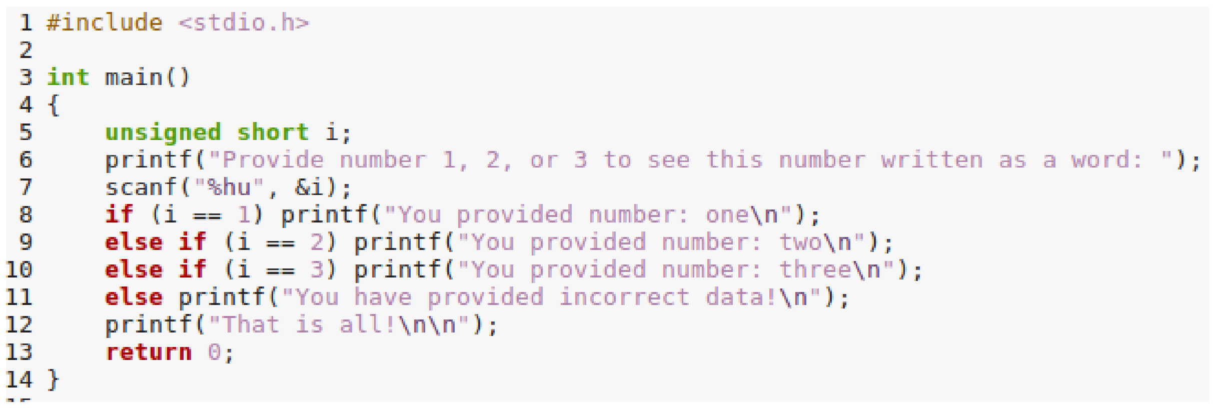
Let's fix these problems. At first, program should continue checking statements until one of them will be fulfilled. If such a statement will be found other statements should be skipped because there is no need to check them anymore. Update a relevant part of the source code of this program. You should achieve result given in Listening 1.3. The second thing is that user can provide other value than 1, 2 or 3. Therefore, an additional `else` option should be added which will be responsible to handle such a case (see Listening 1.5).



```c
#include <stdio.h>

int main()
{
    unsigned short i;
    printf("Provide number 1, 2, or 3 to see this number written as a word: ");
    scanf("%hu", &i);
    if (i == 1) printf("You provided number: one\n");
    else if (i == 2) printf("You provided number: two\n");
    else if (i == 3) printf("You provided number: three\n");
    printf("That is all!\n\n");
    return 0;
}
```

Listing 1.4: Example usage of a conditional statement using the if instruction – update 1



```c
#include <stdio.h>

int main()
{
    unsigned short i;
    printf("Provide number 1, 2, or 3 to see this number written as a word: ");
    scanf("%hu", &i);
    if (i == 1) printf("You provided number: one\n");
    else if (i == 2) printf("You provided number: two\n");
    else if (i == 3) printf("You provided number: three\n");
    else printf("You have provided incorrect data!\n");
    printf("That is all!\n\n");
    return 0;
}
```

Listing 1.5: Example usage of a conditional statement using the if instruction – update 2

Inside the `if else` statement there could be included only one line of source code, like `something_to_do`, or there could be more lines of code. This could be done in case of `true` and in case of `false` as well. More lines of code have to be putted inside a block which is restricted by braces { and }.

1.4 Conditional Operator

Sometimes a simpler version of the `if else` statement is used. It is called a **conditional operator** and it is given as follows:

```
(logical_expression ? in_case_of_true : in_case_of_false)
```

It is commonly used in short and not complicated statements.

1.5 Enumeration Types

Logical values can store only two values, so we can say that we have two options. In case where there is more options needed, however, it is not enough to use logical values. Let's say we have four colors of playing cards, some number of program's modes, options in some system menu, and so on. Each option has its own name and set of such options is generally finite and constant (there are four colors of playing cards, number of program's modes is given by programmer during the development stage, etc.). The easiest way is to give all of these options a particular number, for example:

- $0 \rightarrow$ clubs $\clubsuit$,
- $1 \rightarrow$ diamonds $\diamondsuit$,
- $2 \rightarrow$ hearts $\heartsuit$,
- $3 \rightarrow$ spades $\spadesuit$

and save it in a variable of a proper capacity (for example `unsigned char` which gives 256 possible values and this is enough to storage four values in our case).

Such a solution seems to be right, however, it has some disadvantages. Firstly, such a variable is a normal/typical value and therefore it not differ from other values used in program what could be a problem when some peace of code will be modified later by a person which is not familiar with it or even it will be modified after longer time by a programmer who developed it and who could not remember what a tricky solution he used. Secondly, the compiler will be not able to recognize which values are the proper one and any modifications in future could change these variables' values to any value (even if such a value will be out of expected range/set).

To solve this problem an **enumeration types** are used.

Definition of a new enumeration type is given as follows:

```
enum EnumerationName {
  Position1,
  Position2,
  Position3,
  ...
  PositionN
};
```

And usage in a source code is done as follows:

```
enum EnumerationName MyPosition;
...
MyPosition = Position2;
```

In this case `MyPosition` will has value 1.

1.6 Conditional Construction

If a lot of `if` statements are used in some program to check what what value has some variable (or to choose some menu's option) sometimes is possible to use the `switch` statement. Conditional construction of `switch` is build from four keywords:

- keyword `switch`
- keyword `case`
- keyword `break`
- keyword `default`

A typical construction of `switch` is as follows:

```
switch (some_expression)
{
   case value1:
       ...
       break;
   case value2:
       ...
       break;
   ...
   case valueN:
       ...
       break;
   default:
       ...
       break;
}
```

2 Incrementation and Decrementation

In programs very often are used counters which allows to go step by step in for example loop. Such counters are increased or decreased in each iteration of such a loop. Let's assume that we have counter `counter`. If we want to increase its value we have to provide the following code:

```
counter = counter + 1;
```

In a similar way, when we want to decrease such a counter we have to provide the following code:

```
counter = counter - 1;
```

Both instructions can be simplified using special operators of incrementation or decrementation. To increase some value we can use:

- preincrementation: `++a`
- postincrementation: `a++`

To decrease some value we can use:

- predecrementation: `--a`
- postdecrementation: `a--`

3 Exercises

1. Ask user to give some value.
2. Save this value in a new variable.
3. Display this value in the terminal.
4. Use two incrementation method – display result of their usage in the terminal.
5. Use two decrementation method – display result of their usage in the terminal.
6. Write a new function(s) which will be responsible for drawing a square in the terminal (build from | and -). This function will be drawing a square of size equal to number given by user in a previous task.
7. Let's say user should guess how much apples do you have. Prepare some simple game for this.
8. Draw an algorithm (block diagram) showing how all game is working.