5.2 Writing clean code

The Arduino IDE (Integrated Development Environment) has contributed to clean and readable code by promoting a simple and structured coding approach. One of the ways it achieves this is by introducing the `setup` and `loop` functions.

The "setup" function is called only once when the Arduino board starts up. It is typically used for initializing variables, setting pin modes, and configuring any necessary settings. By separating this initialization code into a designated function, it becomes easier to identify and understand the setup process, making the code more organized and readable.

The "loop" function is called repeatedly after the setup function. This function contains the main logic of the program, where actions are performed continuously. By having a dedicated loop function, it enhances the readability of the code as it clarifies the flow of execution.

The Arduino IDE enforces the presence of these two functions, which serves as a guideline for developers to structure their code. This standardization promotes cleaner code because it encourages encapsulating specific functionalities in separate functions. This helps to compartmentalize different tasks and improves code modularity and maintainability.

Additionally, the Arduino IDE provides various built-in libraries and functions that simplify common tasks, such as reading sensor values or controlling actuators. These libraries follow consistent naming conventions and provide well-documented APIs, making it easier for developers to understand and use them. By leveraging these libraries, developers can write cleaner and more readable code, as they can focus on the high-level functionality and rely on the underlying library for the low-level details.

In order to make your code readable you have to clean your code regularly. This step is very important to not to slow down the programming process in the future programming. You will probably spent the same amount of time cleaning the code that you needed for writing a working version.

In general you can follow some rules:

1. Use FUNCTIONS for every single action,
2. COMMENT the code only where is necessary,
3. Use EXPLANATORY CONSTANTS and VARIABLES

to make your code clean.

Our aim is to write more readable code like in prog. 1:

Program 1: Writing Clean Code.

```
#include "RobotMoves.h"
void setup()
{
  setIOpins();
  moveForward();
  delay(3000);
  stopTheRobot();
}

void loop()
{

}
```

... we will do it in several steps.

5.2.1 Tasks:

1. Write programming functions for moving the robot in several dirrections:
    1. `moveForward()`,
    2. `moveLeft()`,
    3. `moveRight()`,
    4. `moveBackward()`,
    5. `stopTheRobot()`.
2. Save all the functions into header file: `RobotMoves.h`. An example of header file is shown in prog. 2

Program 2: Robot Moves.

```
/***************************
 * IO pins of the Robot
 ***************************/
const int LEFT_MOTOR_PIN_1  = 7;
const int LEFT_MOTOR_PIN_2  = 6;
const int RIGHT_MOTOR_PIN_2 = 5;
const int RIGHT_MOTOR_PIN_1 = 4;
/***************************
 * Function declarations
 ***************************/
void setIOpins();
void moveForward();
/***************************
 * Function definitions
 ***************************/
void setIOpins(){
  pinMode( LEFT_MOTOR_PIN_1, OUTPUT);
  pinMode( LEFT_MOTOR_PIN_2, OUTPUT);
  pinMode(RIGHT_MOTOR_PIN_1, OUTPUT);
  pinMode(RIGHT_MOTOR_PIN_2, OUTPUT);
}
void moveForward(){
  digitalWrite( LEFT_MOTOR_PIN_1, LOW);
  digitalWrite( LEFT_MOTOR_PIN_2, HIGH);
  digitalWrite(RIGHT_MOTOR_PIN_1, LOW);
  digitalWrite(RIGHT_MOTOR_PIN_2, HIGH);
}
```

5.2.2 Questions:

1. Explain why functions contribute to more readable code.
2. Why is good to use explanatory variables?
3. <++>

5.2.3 CLEAN CODE EXPLAINED

5.2.3.1 Comments - YES and NO Comments are very helpful and necessary. Keep them short and meaningful whenever is needed. May also help during thinking process while beginning designing the code.

```
  // robot will go forward
  digitalWrite(7,HIGH);
  digitalWrite(6,LOW);
  digitalWrite(5,HIGH);
  digitalWrite(4,LOW);
```

Don't use comments where the code is self-explanatory, for example:

```
    delay(3000); //wait for 3000ms
```

5.2.3.2 Functions Concatenate programming code into meaningful functions is a must! Previous example of code for `driving a robot forward` is very difficult to understand at first sight. We can make cleaner code as is shown in nex example where is easier to understand what-is-what:

```
  void robotForward()
  {
    digitalWrite(LEFT_MOTOR_PIN_1,HIGH);
    digitalWrite(LEFT_MOTOR_PIN_2,LOW);
    digitalWrite(RIGHT_MOTOR_PIN_1,HIGH);
    digitalWrite(RIGHT_MOTOR_PIN_2,LOW);
  }
```

Compact code is more understandable than large one, see next example:

```
  void setup()
  {
    setIOpins();
    moveForward();
    delay(3000);
    robotStop();
  }
```

5.2.3.2.1 Function declaration Function declaration is highly advisable since allow you a quick overview of available functions in a current file. It is like a table of functions with it's return type and parameters. All declarations are tipically found at the beginig of the file.

```
  void moveForward();
  void moveLeft();
  void moveLeft_PWM(int pwm_value);
```

5.2.3.2.2 Function Definition A function definition provides the actual body of the function.

```
void robotForward()
{
  digitalWrite(LEFT_MOTOR_PIN_1,HIGH);
  digitalWrite(LEFT_MOTOR_PIN_2,LOW);
  digitalWrite(RIGHT_MOTOR_PIN_1,HIGH);
  digitalWrite(RIGHT_MOTOR_PIN_2,LOW);
}
```

5.2.3.3 Constants Use explanatory constants to more clearly represent unintuitive numbers or other abstract values. Use these constants instead of comments since these numbers will appear several times during programming code.

```
const int LEFT_MOTOR_PIN_1 = 7;
const int LEFT_MOTOR_PIN_2 = 6;
```

Now you can easily see why the pins are set as OUTPUT. Because there is `Left motor` attached.

```
void setIOpins()
{
  pinMode(LEFT_MOTOR_PIN_1, OUTPUT);
  pinMode(LEFT_MOTOR_PIN_2, OUTPUT);
}
```

5.2.3.4 Variables Use explanatory variables to make if-statements easily readable and thus understandable. Make `boolean` variables as short statements with no inverting logic.

For example we will use the case where the robot should stop as soon it hits the obstacle with front bumper. The worst case scenario of the program could look like this (we have all done it at some point):

```
void loop()
{
  if (digitalRead(A0) == FALSE){
    digitalWrite(7, HIGH);
    digitalWrite(6, LOW);
    digitalWrite(5, HIGH);
    digitalWrite(4, LOW);
  }else{
    digitalWrite(7, LOW);
    digitalWrite(6, LOW);
    digitalWrite(5, LOW);
    digitalWrite(4, LOW);
  }
}
```

And more clean representation of same functionality is shown in next example of the code. Line 3 is easy readable, simple, clear and easy understandable.

```
1  void loop()
2  {
3      int front_bumper_is_pressed = digitalRead(BUMPER_INPUT);
4      if (front_bumper_is_pressed) robotStop(); else robotForward();
5  }
```

5.2.3.5 Header files To keep our main program file short and transparent as possible we can put supporting code (e.g. functions, settings, ...) into separate file and include it at the beginning of the main program. These files are called header files. We can write a function and save it into header file called “calculate.h”

```
1  int sumTwoNumbers(int A, int B)
2  {
3    return A+B;
4  }
```

In our main program we can include the header file and use the function:

```
1  #include "calculate.h"
2
3  int main()
4  {
5    int a = 5, b = 3;
6    int sum = sumTwoNumbers(a, b);
7  }
```

There are several reasons to use header files in C++:

Code organization: Header files allow you to organize your code into logical units, which can make it easier to understand and maintain. For example, you can use a header file to group together related function declarations, constants, and data types.

Code reuse: Header files can be included in multiple source files, which allows you to reuse the same code in multiple places without having to copy and paste it. This can save time and reduce the risk of errors.

Compilation speed: When you include a header file in a source file, the compiler does not need to recompile the code in the header file every time it compiles the source file. This can significantly improve the compilation speed of your program, especially if the header file contains a large amount of code.

Separation of interface and implementation: Header files can be used to separate the interface (the function declarations and data types that are visible to the rest of the program) from the implementation

(the actual code that performs the tasks). This can make it easier to change the implementation of a module without affecting the rest of the program.

5.2.3.6 Pre-process The preprocessors are the directives, which give instructions to the compiler to pre-process the information before actual compilation starts (e.g. `#include` is one of them). You can easily use as such text substitutions for more clear code reading.

```
#define LEFT_MOTOR_PIN_1 7
#define LEFT_MOTOR_PIN_2 6
```

Remember! `#define` is really a simple text substitution and is not type-safe. Furthermore, we have to be certain that our definition will not interfere with other code used outside of our scope e.g. `libraries`. The last example is not the best representation of `#define` usage. In these case the `const int` is more proper way to go (allowed type checking, debugging). But `#define` has other benefits where `const` can not be used.

5.2.3.6.1 Translations The substitutions can be used as a translation and simplification of code. Such code can be introduced to very young children to get involved in programming.

```
#define vkljuci_led digitalWrite(13,HIGH)
#define izkljuci_led digitalWrite(13, LOW)
#define pocakaj(time) delay(time)
void loop(){
  vkljuci_led;
  pocakaj(1000);
  izkljuci_led;
  pocakaj(1000);
}
```

5.2.3.6.2 Debugging You can even substitute function names e.g. `debug(txt)` with `Serial.println(txt)` and easily separate debugging code lines from necessary serial print of data.

```
#define debug(txt) Serial.porintln(txt)
void setup()
{
  Serial.begin(9600);
  debug("Running...")
}
void loop()
{
  unsigned long myTime = millis();
  Serial.println(myTime);
  delay(1000);
}
```

When we are done with programming and debugging is not needed anymore we can simply change `#define` line to nothing:

```
#define debug(txt)
```

And these programming sentences will not be used. More sophisticated example is shown where programmer can switch between debugging mode (with `#define DBG 1`) and normal operation (with `#define DBG 0`) where code statement `debug("Running...")` will not even compile into program.

```
#define DBG 1
#if DBG == 1
#define debug(txt) Serial.porintln(txt)
#else
#define debug(txt)
#endif
void setup()
{
  Serial.begin(9600);
  debug("Running...")
}
```

5.2.4 Summary:

5.2.5 Issues:

5.2.5.1 What is the difference between `const int` and #define?

`#define` is textual replacement, so it is as fast as it can get. Also it can save some RAM. The downside is that it's not type-safe.

`const` variables may or may not be replaced inline in the code. It is guaranteed to be type-safe though since it carries its own type with it.