

C++

For radio electronics engineers

Vadym Savenko @ 2024

Content:

CHAPTER I. INTRODUCTION TO C++ (p. 3 – 41)

- 1.1. WHY YOU SHOULD LEARN C++ (p. 3 – 7)
- 1.2. VARIABLES AND DATA TYPES (p. 7 – 15)
- 1.3. OPERATORS (p. 16 – 22)
- 1.4. STARVATION (p. 22 – 29)
- 1.5. CYCLES (p. 29 – 32)
- 1.6. FUNCTIONS (p. 32 – 41)
- 1.7. ENABLE USER DATA ENTRY (p. 41)
- 1.8. RANDOMS (p. 41)

CHAPTER II. USING ARDUINO WITH C++ LANGUAGE (p. 42 – 58)

- 2.1. WHAT IS ARDUINO (p. 42)
- 2.2. USING RADIO ELEMENTS WITH ARDUINO (p. 42 – 47)
- 2.3. WHAT YOU NEED TO KNOW WHEN WRITING CODE FOR ARDUINO (p. 47 – 48)
- 2.4. USE OF SENSORS (p. 49 – 56)
- 2.5. USING A LIQUID CRYSTAL SCREEN (p. 57 – 58)
- 2.6. USING THE 7-SEGMENT SCREEN (p. 58)

CHAPTER I. FAMILIARITY WITH C++.

Lesson 1.1. WHY YOU SHOULD LEARN C++.

Great demand

To this day, in various industries, C++ is the preferred choice. Its wide use in existing systems and adaptability to new challenges continue to contribute to its popularity in various industries.

Advantages and Strength of C++:

- **High Productivity:** C++ is known for its high performance and efficient use of system resources.
- **Rich Standard Library:** it has a large standard library that provides a wide range of data structures, algorithms and utilities, saving developers time and effort.
- **Strong Foundation:** provides a deep understanding of computer science, making it an excellent entry point into the world of programming.
- **Skills:** many of the concepts you learn in C++ are applicable to other programming languages, making it easier to learn additional languages in the future.
- **Community support:** It has a large and active community, which means there are a lot of resources, libraries and tools for developers.

C++ is a general-purpose programming language that supports both low-level and high-level programming paradigms, making it a truly adaptable and powerful tool.

- **Low level:** direct hardware control, requires deep understanding of computer architecture, more complex but more accurate.
- **High level:** easier to program, faster to develop, easier to maintain, and has more human-readable code.

| Low level | High level |
|---|---|
| <i>Advantages</i> | |
| full control when making a sandwich from scratch | fast and convenient , because you order a sandwich with optimized cooking |
| <i>Disadvantages</i> | |
| takes a long time and requires skill and experience in making sandwiches | limited customization , lack of direct control over the process of making sandwiches |

Every C++ program must have a function `main()`. Its syntax is as follows:

```
int main()
{
    return 0;
}
```

- `int main()`: starting point of the program. It is called the main function, and it is from it that the execution of the program begins.
- `{ }`: curly braces define a block of code. Everything inside these brackets belongs to the main function and is part of the program logic.
- `return 0;`: marks the end of the program and indicates that it has completed successfully. A value of 0 means everything went well. If there were problems, this value may be different in the output.

Note! Operator `return 0;` is optional at the end of the main function. If you don't insert it, the compiler will add it automatically.

Standard libraries serve as repositories of pre-written, reusable code that simplify common tasks. They save developers time and effort by offering well-tested, standardized tools for building software.

Imagine using the library as a building made of ready-made large blocks. This is much faster and more convenient than creating small blocks from scratch and using them.

Why use standard libraries and files

Writing code with a library is like writing a book with a dictionary. We can easily replace a phrase with just one word without losing the main meaning. Example:

| Text | Another text in which words were replaced by phrases |
|--|--|
| <i>In a biological community, where organisms interact with the physical environment, everything must be in a state in which the various elements are equal or in the right proportions.</i> | <i>In the ecosystem, everything must be in balance</i> |

Preprocessor directives

To add external files to your program, you need to use preprocessor directives. These are commands that control the preprocessor, a tool that transforms code before compilation. The syntax of most preprocessor directives is as follows:

```
#directive parameter
```

- #- symbol indicating that this is a preprocessing directive.
- **directive**: specific preprocessing directive;
- **parameter**: the associated value or argument for this directive.

Team, which adds external files to your program, is called `#include`.

```
#include <name>
```

Note! **Standard files** are attached using angle brackets `< >`, but you can also create your own files and attach them to your project in a similar way using double quotes `" "`.

```
int main()
{
    return 0;
}
```

How `#include` works

Look at the code below and try to run it.

We have two files: `main.cpp`; `header.h`.

Main.cpp code:

```
int main()
{
    return 0;
}
```

Code header.h:

```
}
```

You will get a missing `}` error. This is done on purpose to show how `#include` works. We can create a separate file containing only the `}` character and include it in the `main.cpp` file using the `#include` directive.

The traditional first program in any programming language is often a simple program that outputs a message to the console.

Input and Output

The most common way to output information to the console is to use the `iostream` library. It means input/output flow.

As we already know, to add a library to our program, we must use `#include`.

```
#include <iostream>
```

Character output

`std::cout` is part of the `iostream` library and is used for standard output.

```
std::cout << "Message" << std::endl;
```

```
std::cout << "Message" << std::endl;
```

- `std::` means that the identifier is part of the standard library, which is organized in the `std` namespace.
- `cout` is responsible for character output and is used to send output.
- `<<`: used to insert data into the standard output stream.

Comments

Comments are used for documentation and do not affect program execution. There are two types:

- **Single line comments:** Begin with `//` and extend to the end of the line.
- **Multiline comments:** Conclusions between `/*` and `*/`.

Task. Create a program that outputs your message.

To solve this task, we must use the `iostream` library and use `main()`. You should also use the command to display information:

```
std::cout << "Message";
```

Then we get the code:

```
#include <iostream>
int main()
{
    // Output your message:
    std::cout << "Message";
}
```

But when the code is executed, we get the result in the console:

Message

Lesson 1.2. VARIABLES AND DATA TYPES.

In the realm of programming languages, variables play an important role in storing and manipulating data. The process of creating a variable involves two necessary steps: declaration and initialization.

- **Advertisement:** compiler variable input process. It involves specifying the data type of the variable and optionally giving it a name.
- **Initialization:** the process of assigning a value to a declared variable. This is critical to avoid unexpected or garbage values entering the variable.

```
#include <iostream>

int main()
{
    int x; // Declaration
    x = 25; // Initialization
    std::cout << "My first variable: x = " << x;
}
```

Note! You can specify an initial value for a variable when declaring it.

Also, when declaring several variables that contain data of the same type, it is convenient to specify the data type only once. Example:

```
int myVariable0 = 215;
int myVariable1 = 399;
int myVariable2 = 952;
```

Naming conventions

Choosing meaningful and descriptive variable names is important for writing understandable and maintainable code.

```
int n; // Bad naming
```

OR

```
int kilkistStudentiv; // Nice variable name
```

Variable names in C++ can consist of letters and numbers, but cannot start with numbers, nor can they contain spaces or arithmetic operators. Also, avoid using names that match reserved keywords. This can lead to compilation errors.

You cannot name variables like this in this code:

```
int if = 25; // Uses a reserved word
int email@number = 25; // Contains a special character
int myage = 37; // Contains a pass
int 121A = 121; // Starts with a number
```

When declaring a variable, you need to specify what type of data we will store in it. For ease of working with memory, there are data types for each situation.

| Data Types | |
|-------------|---------------|
| - int | - bool |
| - char | - short |
| - long | - double |
| - long long | - long double |

Numerical

These types are necessary for storing numeric values and manipulating numeric data. We can divide them into two groups:

- Data types for integers (eg 12);
- Data types for floating-point numbers (eg 0.12).

Boolean

Data type `bool` represents two boolean values: zero is interpreted as false and one is interpreted as true.

Light on = true = 1 Light off = false = 0

Char

Typedata `char` is used to store individual characters, which can include letters, numbers, punctuation marks, and special characters.

```
bool isCodefinityCool = true;
char special = '$';
char letter = 'A';
```


Whole numbers

The `int` data type can store values in the range 2,147,483,648 to 2,147,483,647.

```
#include <iostream>

int main()
{
    int goodNumber = 12;
    int tooLarge = 2147483648;

    std::cout<< "Printing tooLarge: " << tooLarge << std::endl;
    std::cout<< "Printing goodNumber: " << goodNumber << std::endl;
}
```

This happens because when you declare a type variable `int`, it allocates exactly 4 bytes of your computer's memory. And numbers above 2147483647 (or below -2147483648) do not fit in these 4 bytes. Fortunately, other available data types can give you more (or less) space for your needs. Here is the table:

| Data type | Range | Size |
|-----------|--|---------|
| short | -32,768 - 32,767 | 2 bytes |
| int | -2,147,483,648 to 2,147,483,647 | 4 bytes |
| long | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 | 8 bytes |

So you can use `long` for storing large numbers (for example, the world's population). You can also use `short`, if you're sure your number won't fall outside the range of -32.768 to 32.767 (for example, to store the user's age). Using `short` will take up less space.

```
#include <iostream>

int main()
{
    short age = 22;
    int likes = 143200;
    long population = 7900000000;
    std::cout<< "Age: " << age << std::endl;
    std::cout<< "Likes: " << likes << std::endl;
    std::cout<< "World's population: " << population << std::endl;
}
```

Note. Be careful with the type of data you choose. If the type range is exceeded, the C++ compiler won't tell you about it, and you'll get an unexpected value without knowing anything is wrong.

Floating point numbers

The above data types are for storing integers. If we try to assign 1.6 to one of them, this is what we get:

```
#include <iostream>

int main()
{
    int num = 1.6;
    std::cout<< num << std::endl;
}
```

Type `int` ignores the decimal part of the number. The same story with `short` or `long`. A data type should be used to store floating-point (decimal) numbers `float` or `double`.

| Data type | Precision | Size |
|-----------|-------------------|---------|
| float | 7 decimal places | 4 bytes |
| double | 15 decimal places | 8 bytes |

Here is a usage example `double` for storage 1.6.

```
#include <iostream>

int main()
{
    double num = 1.6;
    std::cout<< num << std::endl;
}
```

Note. Since the float type has a precision of only 7 digits, the number 123.456789 is already beyond its range. This can lead to inaccurate results (as shown in the example below). Therefore, it is better to use double as the default unless you are sure that the precision of float is sufficient.

```
#include <iostream>

int main()
{
    float floatNum = 123.45678;
    double doubleNum = 123.45678;

    std::cout<< "using float:" << floatNum - 123 << std::endl;
    std::cout<< "using double:" << doubleNum - 123 << std::endl;
}
```

Obviously, you can use float or double to store integers, since they are decimal numbers with a decimal part equal to 0. However, in general, if a variable stores values that can only be integers (such as population or preferences), short/int/long should be used.

```
#include <iostream>

int main()
{
    float price = 100;
    std::cout<< "Price is: " << price << std::endl;;
}
```

You can also store text in variables other than characters using `char`. For this it is necessary

- Attach `string`
- Use scope extension `std`
- Declare a type variable `string`

```
#include <iostream>
#include <string>

int main()
{
    //declaring string variable
    std::string myStringVariable = "codefinity";
    //displaying our string variable
    std::cout<< myStringVariable << std::endl;
}
```

String variables can also contain numbers (in the form of text):

If you try to add two string variables, you get a concatenation (this works without spaces):

```
#include <iostream>
#include <string>

int main()
{
    std::stringvar1 = Hello;// space is also a symbol
    std::stringvar2 = "World";

    //displaying the sum of string variables
    std::cout<< var1 + var2 << std::endl;
}
```

The same will happen with numbers - they will not be added algebraically.

An array is a set of elements of the same type. To create an array, follow these steps:

- Define the data type for the elements you intend to store in the array;
- Assign a name to the array;
- Specify the number of elements in an array by placing that number in square brackets after its name. Example:

```
int myArray[4];
```

The compiler will throw an error if the size is not specified in static arrays.

To initialize the array, you need to specify all its elements in curly brackets:

```
int myArray[5]={-5,423,54,255,1024};
```

To get the element we need from the array, we can refer to it using indexes. Each array element has its own index, just as each house in your city has its own address.

Note.The index starts at index 0, not 1.

| <i>Index</i> | 0 | 1 | 2 | 3 | 4 |
|----------------|----|-----|----|-----|------|
| myArray | -5 | 423 | 54 | 255 | 1024 |

The length of the array above is 6. If we create an array of length 5 with these numbers, it will throw an error.

```
#include <iostream>

int main()
{
    // 1024 is extra element
    int myArray[5]={-5,423,54,6,255,1024};

    std::cout<< myArray[2]<< std::endl;
}
```

Assume that the array has more elements than you declared. In this case, a compilation error** will occur because the compiler allocates a fixed amount of memory when the array is declared. It's like trying to pour more water into an already full glass.

If the array has fewer elements than you specified when declaring it, then all uninitialized elements will be null or have garbage values (unpredictable or random data).

```
#include <iostream>

int main()
{
    int myArray[5]={67,23,87};

    // [3] - index of fourth element
    std::cout<< "My fourth element: " << myArray[3];
}
```

You can think of an array as a book in which each page (element) is numbered (index). The data in the array can be changed, for this you need to refer to the element by index and, for example, set a new value to it:

```
#include <iostream>

int main()
{
    int myArray[3]={67,23,87};

    std::cout<< "my first element: " << myArray[0]<< std::endl;
    std::cout<< "my second element: " << myArray[1]<< std::endl;
    std::cout<< "my third element: " << myArray[2]<< std::endl;

    //change first element
    myArray[0]= -100;
    // Continuation of the code on the next page
```

```
std::cout<< "my first element: " << myArray[0]<< std::endl;
std::cout<< "my second element: " << myArray[1]<< std::endl;
std::cout<< "my third element: " << myArray[2]<< std::endl;
}
```

An array can be an element of another array, for example, let's declare an array whose elements will be other arrays. To declare a multidimensional array, you need another pair of square brackets:

```
int array[][]
```

- The first pair of brackets is the main array;
- The second pair of parentheses indicates that the elements of the main array will be small arrays.

```
#include <iostream>

int main()
{
    //creating multidimensional array
    int myArray[4][3]={ {000,00,0},// first element of main array
{111,11,1},// second element of main array
{222,22,2},// third element of main array
{333,33,3};// fourth element of main array
};

    //display the number 22
    std::cout<< myArray[2][1]<< std::endl;
}
```

We've created an array called myArray that contains four elements, each of which is an array of three elements.

The size of the variable is the amount of memory reserved by the compiler. The compiler reserves a certain number of bytes from your computer's memory based on the data type you are using. You can use the sizeof() function to find the size of a variable or data type in bytes. Example:

```
#include <iostream>

int main()
{
    int myVar1;
    char myVar2;
    std::cout<< "Size of int: " << sizeof(myVar1)<< std::endl;
    std::cout<< "Size of char: " << sizeof(myVar2)<< std::endl;
}
```

C++ allows you to select a type with a precise bit size, such as `int8_t`, `uint8_t`, `int16_t`, `uint16_t`, etc. To use these data types, you need to include the `<cstdint>` header file.

```
#include <cstdint>
```

Alternatively, we can force the compiler to determine the type of the variable itself by using the `auto` keyword.

```
#include <iostream>

int main()
{
    auto myVar = 64.565;

    std::cout<< "Value of myVar : " << myVar << std::endl;

    // double type takes 8 bytes
    std::cout<< "Size of myVar : " << sizeof(myVar)<< std::endl;
}
```

C++ also allows you to rename existing data types to your own. A `typedef` is used for this:

```
#include <iostream>

int main()
{
    //change name of double type to MY_NEW_TYPE
    typedef double MY_NEW_TYPE;

    MY_NEW_TYPE myVar = 64.565;

    std::cout<< "Value of myVar: " << myVar << std::endl;

    // double type takes 8 bytes
    std::cout<< "Size of myVar : " << sizeof(myVar)<< std::endl;
}
```

At compile time, the `typedef` line tells the compiler that `MY_NEW_TYPE` is just a `double` type.

Lesson 1.3. OPERATORS.

Assignment operator (=): In programming, it is used to assign a value to a variable. The syntax is as follows:

```
#include<iostream>

int main()
{
    int myVar = 9;
    int yourVar;

    yourVar = myVar; //assign yourVar the value of myVar

    std::cout<< myVar << std::endl<< yourVar;
}
```

With the string data type, it works exactly the same way:

```
#include <iostream>
#include <string>

int main()
{
    std::string myVar = "codefinity";
    std::string yourVar;

    yourVar = myVar; //assign yourVar the value of myVar

    std::cout<< myVar << std::endl;
    std::cout<< yourVar << std::endl;
}
```

The equality (==) and inequality (!=) operators are used to numerically compare 2 variables:

```
#include <iostream>
int main()
{
    int var = 9;
    int yourVar =(var == 9);
    int myVar =(var == -9);
    std::cout<< yourVar << std::endl;
    std::cout<< myVar << std::endl;}
}
```


Why 1 and 0? This is an alternative approach to using the Boolean data type. When the expression `var == 9` is true, it is represented as 1, which means that `var` is indeed equal to 9. Conversely, when the expression `var == -9` is false, it is represented as 0, which means that `var` is not equal to the number -9.

The inequality operator (`!=`) works exactly the opposite:

```
#include<iostream>

int main()
{
    int var = 9;

    //if var is equal to 9, then 0 (false)
    int yourVar =(var != 9);

    //if var is not equal to -9, then 1 (true)
    int myVar =(var != -9);

    std::cout<< yourVar << std::endl;
    std::cout<< myVar << std::endl;
}
```

These five mathematical operators (+, -, *, / and %) are used to perform various mathematical operations:

```
#include<iostream>

int main()
{
    int myVar = 9;
    int yourVar = 5;

    //using the sum operator (+)
    int resultSum = myVar + yourVar;
    std::cout<< "9 + 5 = " << resultSum << std::endl;
    //using the subtraction operator (-)
    int resultDiff = yourVar - myVar;
    std::cout<< "5 - 9 = " << resultDiff << std::endl;
    //using the multiplication operator(*)
    int resultMult = myVar * yourVar;
    std::cout<< "9 * 5 = " << resultMult << std::endl;
    // Continuation of the code on the next page
```

```
//using the division operator (/)
int resultDiv = myVar / yourVar;
std::cout<< "9 / 5 = " << resultDiv << std::endl;

//using the modulo operator (%)
int resultModDiv = myVar % yourVar;
std::cout<< "9 % 5 = " << resultModDiv << std::endl;
}
```

The division remainder operator (modulo)(%) calculates and returns the remainder of a standard division operation.

The division operator (/) returns only the integer part of the result, discarding the remainder. For example, when dividing 10 by 3, the result will be 3, not 3.333... To get the desired result of division with decimal places (for example, $10 / 3 = 3.333$), it is necessary that at least one of the operands has a double or float data type.

```
#include<iostream>

int main()
{
    // one of the variables must be double or float type
    double myVar = 9;
    int yourVar = 5;

    std::cout<< "9 / 5 = " << myVar / yourVar << " (Expected result)" << std::endl;

    // both operands of integer type
    int myVar1 = 9;
    int yourVar1 = 5;

    std::cout<< "9 / 5 = " << myVar1 / yourVar1 << " (Not expected result)" << std::endl;
}
```

The sum operator is the only mathematical operator that can be applied to a string (it's called concatenation):

```
#include<iostream>
#include<string>

int main()
{
    std::stringmyVar = code;
    std::stringyourVar = "finity";
    // Continuation of the code on the next page
}
```

```
//using sum operator (+) for concatenation
std::string resultSum = myVar + yourVar;
std::cout<< "code + infinity = " << resultSum << std::endl;
}
```

There are also comparison operators (>, <, <=, >=). They are used when you need to numerically compare values to a specific range:

```
#include<iostream>

int main()
{
    int myVar = 9;
    int yourVar = 5;

    bool greater =(myVar > yourVar);
    std::cout<< "9 > 5 is " << greater << std::endl;

    bool greaterOrEqual =(myVar >= myVar);
    std::cout<< "9 >= 9 is " << greaterOrEqual << std::endl;

    bool lessOrEqual =(myVar <= yourVar);
    std::cout<< "9 <= 5 is " << lessOrEqual << std::endl;

    bool less =(myVar < yourVar);
    std::cout<< "9 < 5 is " << less << std::endl;
}
```

To evaluate multiple conditions at once, you must use logical NOT operators (!), AND (&&) and OR (||). Let's illustrate the use of logical operators in real-world scenarios:

- I'll take my wallet on a walk if there's a bank AND a store on my way;
- I will not take my wallet for a walk if there is NOT a bank on the way;
- I will take my wallet on a walk if there is a bank OR store along the way.

```
#include <iostream>

int main()
{
    bool bank = true;
    bool shop = true;
    //using AND (&&) operator
    bool wallet1 =(bank && shop);
    std::cout<< "There is a bank and a shop, wallet = " << wallet1 << std::endl;
    // Continuation of the code on the next page
}
```

```
//using NOT (!) operator
bool wallet0 =(!bank);
std::cout<< "There is no bank, wallet = " << wallet0 << std::endl;

//using OR (||) operator
bool wallet2 =(bank || shop);
std::cout<< "There is a bank or a wallet = " << wallet2 << std::endl;
}
```

Compound assignment operator is a combination operator that combines an assignment operator and an arithmetic (or bitwise) operator. The result of such an operation is assigned to the left operand (the one before the = sign).

```
#include <iostream>

int main()
{
    int some_variable = 0;
    std::cout<< "Old value of the variable: " << some_variable << std::endl;

    some_variable += 500; // using a compound assignment operator
    std::cout<< "New value of the variable: " << some_variable << std::endl;
}
```

Using another compound assignment operator:

```
#include <iostream>

int main()
{
    int value = 0;

    value += 20; // value = value + 20
    std::cout<< "value = 0 + 20 = " << value << std::endl;

    value -= 4; // value = value - 4
    std::cout<< "value = 20 - 4 = " << value << std::endl;
}
```

```

value *= 4; // value = value * 4
std::cout<< "value = 16 * 4 = " << value << std::endl;

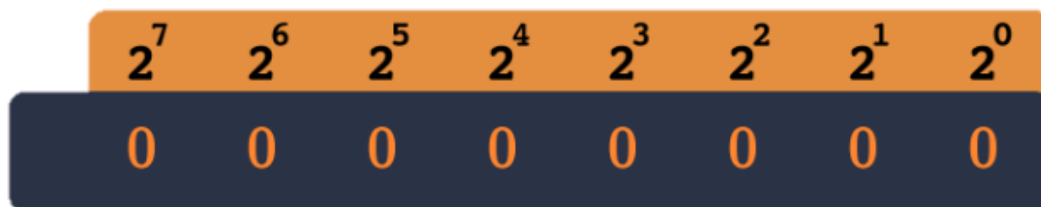
value /= 8; // value = value / 8
std::cout<< "value = 64 / 8 = " << value << std::endl;

value %= 5; // value = value % 5
std::cout<< "value = 8 % 5 = " << value << std::endl;
}

```

What is a byte?

This is a byte, the fundamental unit of digital information storage. A byte consists of 8 bits.



The basis of digital information storage is a byte consisting of 8 bits. Each bit represents a digit, and numbers are usually expressed in binary. Binary representation involves expressing a number as a sum of powers of 2.

Examples of binary representation

$$1010 = 10$$

$$111111 = 255$$

$$101010 = 42$$

Consider the number 4, which in the binary number system has the form 00000100. Each bit in this byte corresponds to a power of 2:

$$0+0+0+0+0+2^2+0+0 = 4$$

Similarly, the number 5 is represented as 00000101:

$$0+0+0+0+0+2^2+0+2^0 = 5$$

Bitwise shift

Bitwise shift

- In this context, one shift to the left: the value multiplies the value by 2.
- Conversely, when we perform a one-bit shift to the right: the value is divided by 2.

```
#include <iostream>

int main()
{
    int value = 20;
    std::cout<<(value << 1)<< std::endl;
}
```

Think of the operation of shifting right and left \gg OR \ll by n positions as dividing or multiplying by 2^n .

In the case of $10 \gg 3$, this is actually dividing 10 by 2^3 (which is 8), and the result is 1.

Mathematical operators needed for the following problem

| Operator | Description |
|----------|--|
| - | Subtracts one value from another |
| * | Multiplies two values |
| / | Divides one value by another |
| + | Adds one value to another |
| % | Shows the remainder when dividing one value by another |
| ** | Raise to a power |
| // | Target division |

Lesson 1.4. STARVATION.

The if...else construct in programming allows your program to choose different paths and manage different potential outcomes.

It consists of two main components: a condition and corresponding actions or consequences based on this condition.

Here is an illustration:

```
#include<iostream>

int main()
{ // Continuation of the code on the next page
```

```

int var = 13;

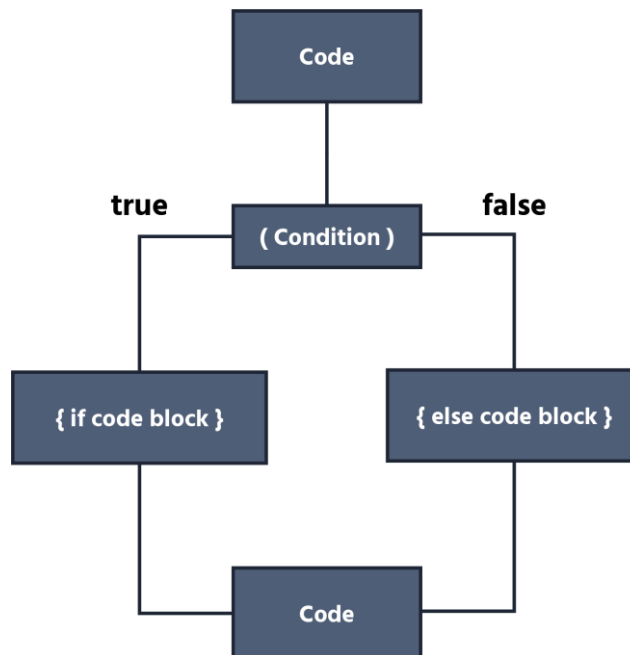
/* If my variable equals 13,
then print "OKAY", and
change variable to 15 */

if(var == 13)
{
    std::cout<< "13 == 13, it is OKAY" << std::endl;
    var = 15;
}

/* New value of variable (15) doesn't equal 13,
then print "NOT OKAY" */

if(var != 13)
{
    std::cout<< "15 != 13, it is NOT OKAY" << std::endl;
}
}

```



There is also handling of the opposite case using else:

```

#include<iostream>

int main()
{
    int var = 200;
    // Continuation of the code on the next page
}

```

```

/* If my variable equals 13,
then print "OKAY" */

if(var == 13)
{
    std::cout<< "Variable is 13, it's OK" << std::endl;
}

/* If my variable doesn't equal 13,
then print "NOT OKAY" */

else
{
    std::cout<< "Variable isn't 13, it's NOT OK" << std::endl;
}
}

```

Inside if...else there can be other if...else:

```

#include<iostream>

int main()
{
    int var = 15;

    if(var == 15)
    {
        //then
        var = 15 + 200;

        if(var == 300)
        {
            //then
            std::cout<< "Okay" << std::endl;
        }

        // otherwise
        else
        {
            std::cout<< "NOT OKAY" << std::endl;
        }
    }
}

```


You can also use the else if construct:

```
#include<iostream>

int main()
{
    int var = 50;

    if(var == 15)
    {
        std::cout<< "var equals 15" << std::endl;
    }
    else if(var == 50)
    {
        std::cout<< "var equals 50" << std::endl;
    }
    else if(var == 89)
    {
        std::cout<< "var equals 89" << std::endl;
    }
    else if(var == 215)
    {
        std::cout<< "var equals 215" << std::endl;
    }
}
```

The operator `if...else` offers a concise alternative to the operator `if...else`, with a noticeable difference. It consists of three key elements:

1. Boolean expression;
2. Case instruction `true`;
3. Case instruction `false`.

```
(booleanexpression)?instruction_for_the_true_case:instruction_for_false_case
```

You can also do this to get the result in the console:

```
#include <iostream>
using namespace std;

int main()
{ //continuation of the code on the next page
```

```
string a = "Hello, World!";  
cout << a;  
}
```

Such an operator is convenient to use, for example, when comparing two numbers:

```
#include <iostream>  
  
int main()  
{  
    int var1 = 50;  
    int var2 = 9;  
  
    int result =(var1 > var2)? var1 : var2;  
  
    std::cout<< result << std::endl;  
}
```

In this case, the result of the triple operation was assigned to the variable result.

When the comparison returns true, the value of var1 will be stored in the result variable.

if true then

int result = (var1 > var2) ? var1 : var2;



Conversely, if the result of the comparison is false, the variable result will be assigned the value of the variable var2.

if false then

int result = (var1 > var2) ? var1 : var2;



Note.Observe data type compatibility!

How would it look using if...else:

```
#include<iostream>

int main()
{
    int var1 = 50;
    int var2 = 9;
    int result;

    if(var1 > var2)
    {
        result = var1;
    }
    else
    {
        result = var2;
    }

    std::cout<< result << " > " << var2 << std::endl;
}
```

The switch-case construct allows you to compare the result of an expression with a set of predefined values. Switch-case structure:

```
switch(someExpression)
{
    case someCheck1:
        // if this case is allowed
        do_something1;
        break;

    case someCheck2:
        // if this case is allowed
        do_something2;
        break;

    Default:
        //if none of the cases match
        do_something_else;
}
```

```
#include <iostream>

int main()
{
    int variable = 5;

    // as the expression to be checked, we will simply have our variable
    switch(variable)
    {
        case 5://if variable equals 5
        std::cout<< "Value of variable equals 5" << std::endl;
        break;

        case 20://if variable equals 20
        std::cout<< "Value of variable equals 20" << std::endl;
        break;
    }
}
```

- break- is an operator that means exit from a block of code;
- default- this is an optional, but useful part. This part will be executed if none of the options are suitable.

In our case, we check `variable`, if it is equal to 5, then the corresponding text will be displayed, and using the operator `break`, the program flow will exit the entire `switch-case`, and no other cases will be processed.

- break- operator means exit from the block of code.
- default- optional, but useful part. This part will be executed if none of the cases are suitable.

In our case, we check `variable`, if it is equal to 5, then the corresponding text is displayed using the operator `break` the program flow will leave the entire `switch-case`, and there will be no processing of other cases.

```
#include <iostream>

int main()
{
    int variable = 5;
    switch(variable)
    {
        case 5:
        std::cout<< "Value of variable equals 5" << std::endl;
        // delete "break;"
        // Continuation of the code on the next page
    }
}
```

```

case 10:
std::cout<< "Value of variable equals 10" << std::endl;
// delete "break;"

case 15:
std::cout<< "Value of variable equals 15" << std::endl;
// delete "break;"
}
}

```

But the switch statement has one caveat. We intentionally remove the break statement:

We used if...else, switch-case to compare our variables with other values. But what if we need to do something a hundred times? A thousand times? A million times?

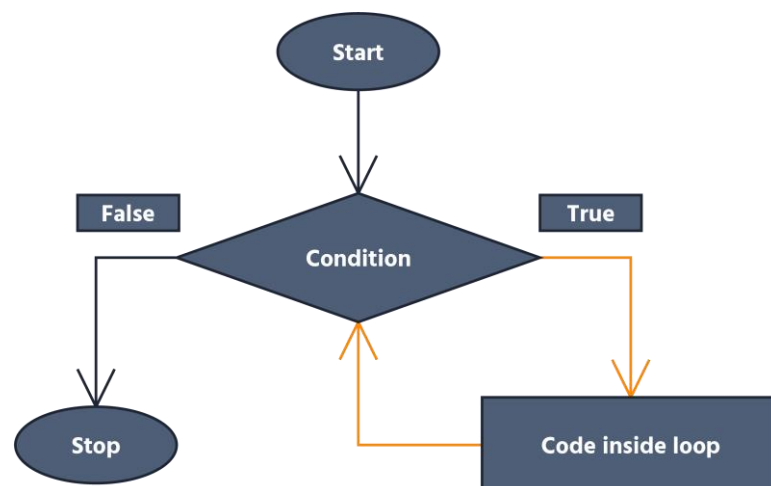
Cycles are designed for such cases! They allow you to loop your program under certain conditions. The structure of the while loop:

Lesson 1.5. CYCLES.

```

while(someExpression== true)
{
// if someExpression == true, then do_something;
}

```



```

#include <iostream>

int main()
{
    //x + y = result
    int x = 0; //root of equation
    int y = 8;
    int result = 1000;

    //increase x, until it satisfies the equation
    while(y + x != result)
    {
        x += 1; //x = x + 1
    }

    std::cout << "Root of the equation: " << x;
}

```

In this case, we counted (x+=1) 992 times. The loop was executed until x+y equaled the result (1000).

As soon as the expression $x + y$ became equal to the result, the cycle ended, and we got the root of the equation (x).

Note. If the condition is not met, the loop may not start.

It is important to make sure that the loop has an exit condition, that is, that the loop will not be infinite. An example of an infinite loop:

```

#include <iostream>

int main()
{
    bool condition = true;

    while(condition)
    {
        std::cout << "Loop is infinite!" << std::endl;
    }
}

```

Unlike a while loop, which may never execute, a do...while loop is guaranteed to execute at least once. The structure of the do...while loop:

```
do
{
//do_something;
}
while(someExpression== true);
```

Note. The line containing the while part ends with a semicolon (;).

The for loop is more complex than other loops and consists of three parts. The structure of the for loop:

```
for(counter; exit condition; loop expression)
{
// instruction block
do_something;
}
```

- Counter;
- Exit condition;
- Loop expression.

```
#include <iostream>

int main()
{
    for(int counter = 0; counter <= 5; counter++)
    {
        std::cout<< counter << std::endl;
    }
}
```

- `int counter = 0`: iteration counter;
- `counter++`: For each iteration, 1 will be added to the `counter` variable to mark the passage of the loop;
- `counter <= 5`: loop termination condition. The loop will continue if the `counter` variable is less than or equal to 5.

Now the question is, How many iterations will this loop do?:

```
for(int i= -5; i<= 0; i++)
```

A) 5; B) 6; C) 1; D) 0;

Lesson 1.6. FUNCTIONS.

Functions are small routines that can be called as needed. Each function has a name by which it can be called.

```
int main()// "main" is the name of the function
{
    return 0
}
```

Note. The name main is already reserved in C++. Therefore, when declaring a function with such a name, the compiler will throw an error.

To create a function, you need:

- determine the type of data it will return;
- give her a name;
- enclose a block of instructions (body) in curly braces { . . . } to determine its functionality.

For example, let's create a function that outputs the text "c<>definity":

For example, let's create a function that outputs text "c<>definity":

```
std::string nameOfCourses()// function type and name
{// start of body

    std::string nameOfCourse = "c<>definity";
    return nameOfCourse;

} // end of body
```



```

#include <iostream>
#include <string>

std::string nameOfCourses()// type and name of function
{// beginning of a body

    std::string nameOfCourse = "c<>definity";
    return nameOfCourse;

}// end of a body

int main()
{
    std::cout<< "Name of course: " << nameOfCourses()<< std::endl;
}

```

Let's create a function that simplifies the process of converting temperatures from Fahrenheit to Celsius. This is a practical application in everyday life.

```

#include <iostream>

int FahrenheitToCelsius(int degree)
{
    int celsius =(degree - 32)/ 1.8;
    return celsius;
}

int main()
{
    int Fahr = 80;
    std::cout<< Fahr << "F" << " = " <<
    FahrenheitToCelsius(Fahr)<< "C" << std::endl;
}

```

Note!The function argument is represented by the degree variable, which contains the data the function works with. In this context, it refers to temperatures in degrees Fahrenheit that need to be converted to degrees Celsius. We'll take a closer look at function arguments later.

The compiler processes our program code sequentially, just like a human reading a book, and if it encounters unknown variable or function names, it will throw an error.

For example, let's try to call a function before its definition.

This example throws an error. This is done on purpose.

```
#include <iostream>

int main()
{
    intFahr= 80;
    std::cout<<Fahr<< "F" << " = " << FahrenheitToCelsius(fahr)<< "C" << std::endl;
}

int FahrenheitToCelsius(int degree)//creating function AFTER it calling
{
    intcelsius=(degree- 32)/ 1.8;
    returncentigrade;
}
```

In such situations, it is important to use function prototypes.

The purpose of a prototype is to tell the compiler about our function in advance. Creating a prototype is similar to a standard function declaration, but with a subtle difference:

- specify the type of future function;
- give her a name;
- **arguments**(if necessary);
- mark the end of the expression.

```
int FahrenheitToCelsius(int degree);//prototype of our function
```

Let's add a function prototype to our example that caused the error:

```
int FahrenheitToCelsius(int degree);//prototype of our function
```

Note!Prototyping is useful when you are working with a lot of functionality. To avoid "garbage" in the main file, prototypes and function definitions are transferred to third-party files and connected to the main file using the #include directive.

```

#include <iostream>

int FahrenheitToCelsius(int degree);

int main()
{
    int Fahr = 80;
    std::cout<< Fahr << "F" << " = " <<
    FahrenheitToCelsius(Fahr)<< "C" << std::endl;
}

int FahrenheitToCelsius(int degree)
{
    int celsius =(degree - 32)/ 1.8;
    return celsius;
}

```

Note! Prototyping is useful when you are working with a large number of features. To avoid "garbage" in the main file, prototypes and function definitions are transferred to third-party files and included in the main file using the #include directive.

When you create a function, you always specify the type of data it returns.

In the example of the main function, we can see that it has an integer data type, which means that it will return an integer value, in our case the number 0, when it is finished.

```

int main()
{
    return 0;
}

```

Note. Since the main function is reserved in C++, it will always return an integer.

But our functions can return any value:

```

double notMainFunc()
{
    return 3.14;
}

```

To call a function, you need to write its name in parentheses:

```
notMainFunc();
```

```
#include <iostream>

double notMainFunc()
{
    return 3.14;
}

int main()
{
    std::cout<< notMainFunc();
}
```

We created a function that returns the value 3.14 as a double data type, and we called this function to display its output on the screen.

Functions can also be string type:

```
#include <iostream>
#include <string>
std::string notMainFunc()//string function
{
    return "codefinity";
}

int main()
{
    std::cout<< notMainFunc();//calling string function
}
```

typedef can also be used:

```
#include <iostream>

typedef int MY_NEW_TYPE;

MY_NEW_TYPE TYPEfunc()//MY_NEW_TYPE function
{
    return 777;//continuation of the code on the next page
}
```

```

}

int main()
{
    std::cout<< "New type func returned " << TYPEfunc()<<
    std::endl;
}

```

If you cannot specify the return type exactly, the auto statement will make the compiler do it for you:

```

#include <iostream>

auto autoFunc1()// first auto-type function
{
    return 777;
}

auto autoFunc2()// second auto-type function
{
    return 123.412;
}

int main()
{
    std::cout<< "First function returned" << autoFunc1()<< std::endl;
    std::cout<< "Second function returned" << autoFunc2()<< std::endl;
}

```

The return statement completes the execution of the function and returns a value of the specified type.

```

int func()//int - predefined
{
    variableint = 10;
    returnvariable;//variable = 10
}

```

If the type is specified incorrectly, the function will behave unpredictably:

```

#include <iostream>

unsigned short func()
{
    return -10;
}// Continuation of the code on the next page

```

```
//The unsigned short data type has no negative values.

int main()
{
    std::cout<< func()<< std::endl;
}
```

That is, before creating a function, it is necessary to specify the type of data that it returns. There are also special void functions in C++. Functions with this data type may return nothing or "nothing".

```
#include <iostream>

void voidFunction()
{
    std::cout<< "It's void function!" << std::endl;

    //function without return
}

int main()
{
    voidFunction();
}
```

```
#include <iostream>

void voidFunction()
{
    std::cout<< "It's void function!" << std::endl;
    return;
}

int main()
{
    voidFunction();
}
```

Note. Usually, functions of the void type simply output static text or work with pointers

```
#include <iostream>

int func()
{
    int a = 50;
    int b = 6;

    return a;
    return b;
}

int main()
{
    std::cout<< func()<< std::endl;//func calling
}
```

A function with a parameter (argument) is a function that must work with the object "from the outside".

```
int func(int argument)
{
    return argument;
}
```

Note. A function argument is a local variable that is created and exists only in the current function.

For example, let's create a function that divides any integer by 2:

```
int func(int argument)
{
    int result =argument/ 2;
    return result;
}
```

To use such a function, you need to call it and pass it an argument:

```
func(10);
```

Let's give an example:

```
int func(int argument)
{
    return argument;
}
```

Several arguments can be passed to the function:

For example, let's create a function that divides any integer by 2:

`func(5, 7)` - calling the function and passing arguments to it. Arguments are passed through commas (,).

You can also pass arrays:

To use such a function, you must call it and pass it an argument:

```
func(10);
```

Here is an example:

```
#include <iostream>

int func(int argument)
{
    int result=argument/ 2;
    return result;
}

int main()
{
    //function calling and passing it an argument
    std::cout<< "func() returned: " << func(10);
}
```

You can pass multiple arguments to a function:

```
#include <iostream>

int func(int a,int b)
{
    return a+b; //the function to sum arguments
}

int main()
{
    std::cout<< "sums the arguments = " << func(5,7);
}
```

`func(5, 7)` – calling a function and passing arguments to it. Arguments are passed through commas (,). You can also pass arrays:


```
#include <iostream>

//the size of the passed array is optional
int func(int arrayForFunc[])
{
    return arrayForFunc[2]; //function will return third element
}

int main()
{
    int array[6]={75,234,89,12,-67,2543};

    //calling function
    std::cout<< "Third element of array is: " << func(array);
}
```

Lesson 1.7. ENABLE USER DATA ENTRY.

If we know about `std::cout` to display text, but we can still ask the user to write data to a variable using the `std::cin` command. Such code looks like this:

```
#include <iostream>

int main()
{
    std::string text;
    std::cout<< "Text: " << std::endl;
    std::cin>>text;
    std::cout<<text<< std::endl;
}
```

Lesson 1.8. RANDOM

To find a random number that the computer will generate, you need to randomize the numbers in the code. Such code looks like this:

```
#include <iostream>

int main()
{
    int n=/*From what number will be random*/;
    int m=/*Until what number will be random*/;

    /*Finding a random number*/
    int random=n+ rand()%m;
    /*Show a random number in the console*/
    std::cout<<random<< std::endl;
}
```

CHAPTER II. USING ARDUINO WITH C++ LANGUAGE.

Lesson 2.1. WHAT IS ARDUINO.

Arduino (Arduino) is a hardware computing platform for amateur construction, the main components of which are a microcontroller board with input/output elements and the Processing/Wiring development environment in the programming language, which is a simplified subset of C/C++. Arduino can be used both to create autonomous interactive objects and to connect to software running on a computer (for example: Processing, Adobe Flash, Max/MSP, Pure Data, SuperCollider). Information about the board (picture of the printed circuit board, specifications of the elements, software) are in the public domain and can be used by those who prefer to create boards with their own hands.

The name Arduino comes from a bar in Ivrea, Italy, where some of the founders of the project met. The bar was named after Arduin I, who was Margrave of the Marches of Ivrea and King of Italy from 1002 to 1014.

Lesson 2.2. USING RADIO ELEMENTS WITH ARDUINO.

To learn how to use radio elements with Arduino, you must first learn about using radio elements without Arduino.

Radio electronics is a branch of science and technology that covers the theory, methods of creating and using devices for transmitting, receiving and converting information using electromagnetic energy.

The term "radio electronics" appeared in the 50s of the 20th century and is somewhat conditional. Radio electronics covers radio engineering and electronics, including semiconductor electronics, microelectronics, quantum electronics, IR engineering, chemotronics, optoelectronics, acoustoelectronics, cryoelectronics and other radio electronics is closely related, on the one hand, to radio physics, solid state physics, optics and mechanics, and on the other hand, with electrical engineering, automation, telemechanics and computing.

Methods and means of radio electronics are widely used in radio communication, space technology, remote control systems, radio navigation, automation, computer technology, radar, military equipment and special equipment, household appliances, etc. The product of radio electronics is radio electronic equipment.

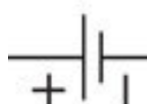
The field of use of radio electronics is continuously expanding, penetrating the economy, industrial production, agriculture, medicine, transport and other areas of human activity.

A power source is required to provide a signal in electronics. In general, the power source can be: a battery; battery; plug with wire; etc. Each power supply has charges such as "Anode" and "Cathode". The cathode is the electrode of some device connected to the negative pole of the current source, and the anode is the electrode from which the current is directed in the gas or electrolyte (the movement of positive charges).

In the literature, there are different designations of the cathode sign - "—" or "+", which is determined, in particular, by the features of the described processes. In electrochemistry, it is generally accepted that the "—" cathode is the electrode where the reduction process takes place, and the "+" anode is the one where the oxidation process takes place. During operation of the electrolyzer (for example, during copper refining), an external current source provides an excess of electrons (negative charge) on one of the electrodes, the metal is restored here, this is the cathode. The lack of electrons and oxidation of the metal is ensured at the other electrode, this is the anode. However, during the operation of a galvanic cell (for example, a copper-zinc one), an excess of electrons (and a negative charge) on one of the electrodes is provided not by an external current source, but by the metal oxidation reaction (zinc dissolution), that is, in the galvanic cell negative, if you follow the given definition, will be the anode. Electrons, passing through the outer circuit, are spent on the reduction reaction (copper), that is, the positive electrode will be the cathode. So, in the given illustration, the cathode of a galvanic cell, marked with a "+" sign, on which copper is reduced, is depicted. According to this interpretation, for the battery, the sign of the anode and cathode changes depending on the direction of current flow.

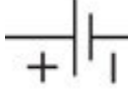




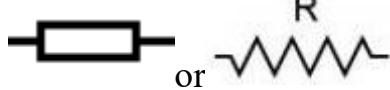


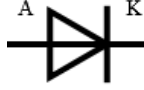

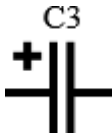
In electrical engineering, the direction of current is considered to be the direction of movement of positive charges, therefore, in vacuum and semiconductor devices and electrolysis cells, the current flows from the positive anode to the negative cathode, and electrons, accordingly, vice versa, from the cathode to the anode.

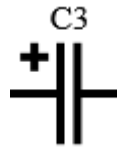





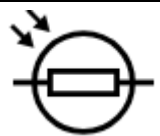
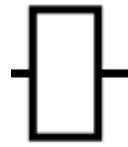


In general, about the creation of radio equipment, a schematic of the device is always made. This is how the power source is designated:



In electronics, elements are divided into information input (Such as; resistors; diodes; sensors; etc.) and information output (Such as: LEDs; buzzer; motor; etc.). An LED is an element that works like a light bulb. A buzzer is an element that makes an audible noise.

Here is a table of how the elements are designated:

| Element | Schematic image |
|--------------------------------------|---|
| Power supply |  |
| LED |  |
| Boozer |  |
| Dynamic |  |
| Button |  |
| Constant resistor |  |
| Variable resistor (Potentiometer) |  |
| Resistor with indication of power |  |
| Diode |  |
| Zener diode |  |
| Capacitor |  |

| | |
|-----------------|--|
| Polar capacitor |  |
| Switch |  |
| Engine |  |
| Inductor |  |
| Voltmeter |  |
| Ammeter |  |
| Photoresistor |  |
| Relay |  |
| NPN transistor |  |
| PNP transistor |  |

A tact button (sometimes called a "push button") is a switch that, when pressed, closes or opens an electrical circuit. It has two states: closed (when pressed) and open (when not pressed). It is used to start or stop various electronic devices or to execute certain commands.

A resistor is a component that limits current in an electrical circuit. It has a certain resistance, which is measured in ohms (Ω). Resistors are used to control current, distribute voltage and protect other components from excessive current.

A potentiometer is a variable resistor whose resistance can be adjusted manually. Potentiometers are often used as volume controls in audio equipment or to adjust various parameters in electronic circuits.

A photoresistor (or LDR - Light Dependent Resistor) is a resistor whose resistance changes depending on the intensity of light. It is used in light sensors and various automatic systems where a reaction to changes in light is required.

A diode is a semiconductor component that allows current to flow in one direction only. It is used to rectify current, protect circuits from reverse polarity, and for various other purposes in electronic devices.

Zener diode is a special type of diode that stabilizes the voltage at a certain level. It is used in circuits to provide a stable supply voltage or to protect against voltage fluctuations.

A capacitor is a component that stores an electrical charge. It has two covers separated by an insulator (dielectric). Capacitors are used for signal filtering, ripple reduction, energy storage, and other purposes.

A polar capacitor is a type of capacitor that has polarity: positive and negative terminals. It has a higher capacity compared to non-polarized capacitors, but requires the correct polarity of connection. It is used in direct current circuits.

An ammeter is a measuring device for measuring current in an electric circuit. Its measurement units are amperes (A). The ammeter is connected in series to the circuit.

A voltmeter is a measuring device for measuring the voltage in an electric circuit. Voltage is measured in volts (V). The voltmeter is connected in parallel to the circuit components.

An NPN transistor is a type of bipolar transistor where two layers of "n" type semiconductor material are surrounded by one layer of "p" type. It is often used in circuits to amplify or switch signals.

A PNP transistor is a bipolar transistor in which two layers of p-type semiconductor material are surrounded by an n-type layer. PNP transistors are also used to amplify and switch signals, but work in reverse mode compared to NPN transistors.

A power-rated resistor is a resistor that is rated for the maximum power it can withstand without overheating and damage. Power is measured in watts (W). Choosing a resistor with the appropriate power is important to prevent it from overheating and failing in the circuit.

For further study of radio electronics, we advise you to read the books "R. Tokheim: Fundamentals of digital electronics" and "P. Horowitz, W. Hill: The Art of Circuit Engineering"

Lesson 2.3. WHAT YOU NEED TO KNOW WHEN WRITING CODE FOR ARDUINO.

In general, when writing code for Arduino, we use the "sketch.ino" file, which has the following code:

```
void setup() {  
  // function setup()  
}  
  
void loop() {  
  // function loop()  
}
```

The setup() function executes all the commands it has when the Arduino is started (In general, these can be commands for setting up elements such as: liquid crystal display; servo; DC motor; etc.), and the loop() function of the Arduino continuously executes all teams that he has.

You should also remember the commands used for Arduino. Here is the code with the basic commands in the loop() function to run the commands continuously:

```
void loop() {  
  /*Output text value in the console.  
  You can output English letters*/  
  Serial.println("Hello World!");  
  
  //Timer to wait time in milliseconds  
  delay(1000); // Wait 1 second  
  
  //Set pin "13" to "HIGH"  
  digitalWrite(13, HIGH);  
  
  //Set pin "13" to "LOW"  
  digitalWrite(13, LOW);  
  /*Outputs in Arduino:
```

```

Analog pins: A0; A1; A2; A3; A4; A5;
Digital pins: 0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13;
Power: 3.3V; 5V;
Grounding: GND;

Analog is continuous communication (For example: light; sound),
while digital is used to read and transmit a digital signal, such
as "0" or "1". */

inta= 0;
intb= 0;
intc= 0;
intd= 0;

// Condition if both comparison operators are "True"
if(a==b&& c==d) {
...
}else{
...
}

// Condition if at least the comparison operator is "True"
if(a==b&& c==d) {
...
}else{
...
}
}

```

When a circuit is created using Arduino, it is marked on the circuit as follows:

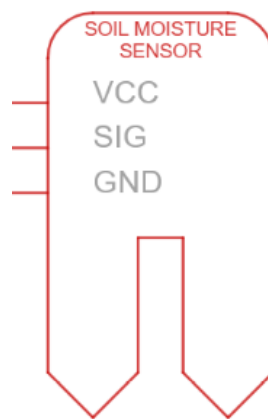


When a circuit is created using Arduino, it is marked on the circuit as follows:

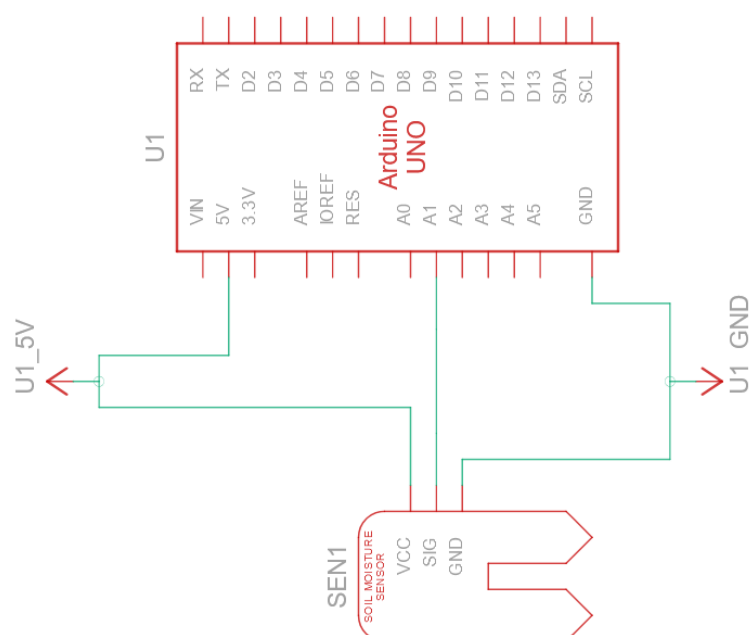
Lesson 2.4. USE OF SENSORS.

Sensors are elements of radio electronics that receive a signal from something. For example: a gas sensor receives a signal due to gas; the humidity sensor receives a signal due to water and humidity; photoresistor (sensor of presence of light) receives a signal from the illumination of the surrounding world; etc. In general, sensors have 3 outputs: power supply (VCC); grounding (GND); signal (SIG). There are also resistors, such as: constant resistors; potentiometers; photoresistors; etc., which only need power and have 2 outputs, where one has power and the other has power and we get a signal.

First, let's deal with the humidity sensor. The humidity sensor serves to receive a signal from water and room humidity. This is how it is indicated in the diagrams:



Let's make an Arduino code that will send humidity information to the console. Here is his scheme:



And here is its code:

```
int moisture = 0;

void setup()
{
  pinMode(A1, INPUT);
  Serial.begin(9600);
}

void loop()
{
  moisture = analogRead(A1); // We get the result
  // humidity in variable
  Serial.println(moisture);
}
```

Now let's move on to the temperature sensor. The thermistor (temperature sensor) serves to obtain information about the temperature in the environment. When the environment is heated, the thermistor voltage increases. There are many types of temperature sensors, such as: TMP36; LM35; etc. But the best temperature sensor is the TMP36, because it can detect both sub-zero and over-zero temperatures, because it receives analog value size data. The formulas for finding degrees Celsius and voltage from the TMP36 analog input are at the beginning of the next page.

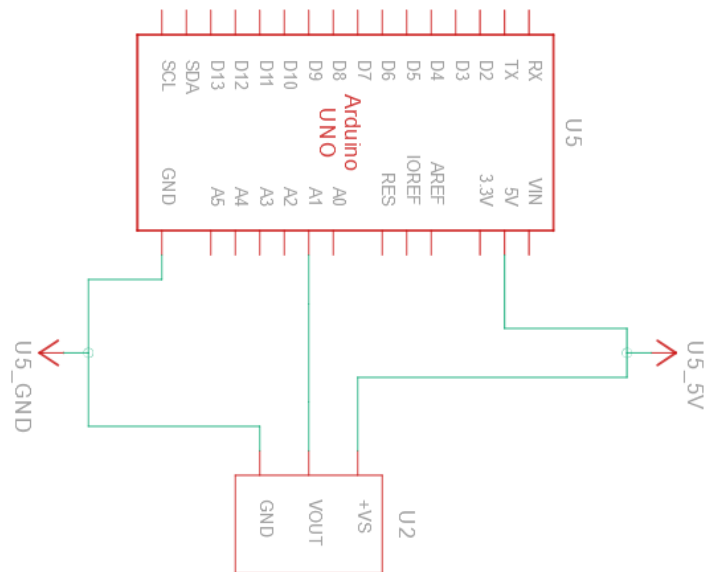
$$\text{Напруга (Вольт)} = \frac{(\text{Аналоговe значення}) * 1.1}{1024}$$

$$\text{Температура (Цельсіях)} = (\text{Напруга} - 0.5) * 100$$

If the TMP36 received a signal at 50 degrees Celsius, it will output 1 volt. This is how TMP36 is marked on the diagram:



Now let's make a code that will display information about volts and temperature in degrees Celsius from the thermistor in the console. Here is the scheme:



Here is the circuit code:

```
void setup()
{
  Serial.begin(9600);
  analogReference(INTERNAL);
}

void loop()
{
  // Continuation of the code on the next page
  int reading= analogRead(A1);
  float voltage=(reading* 1.1)/ 1024.0;
  float temperatureC=(voltage- 0.5)* 100;

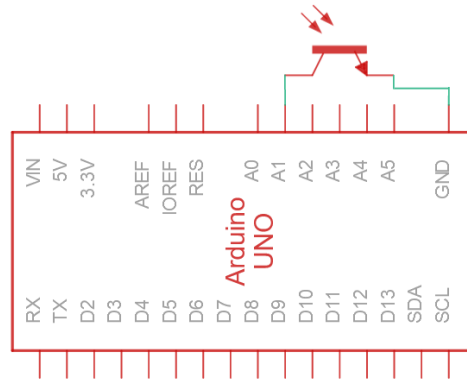
  // First page
  Serial.print(voltage);
  Serial.println("volts");

  // Second page
  Serial.print(temperatureC);
  Serial.println("dagees C");
}
```

There are also light sensors that serve to issue a signal about the lighting of the space. The light sensor is a semiconductor element, the resistance of which changes when the light rays change. When there is lighting, it passes the voltage. This is how it is indicated in the diagrams:



Now let's make a code in which the light sensor, if there is a presence of light rays, we get "1" in the console, and "0" otherwise. Here is the diagram:



Here is the code:

```
void setup()
{
  pinMode(A1, INPUT);
  Serial.begin(9600);
}

// Continuation of the code on the next page
void loop()
{
  // We get the humidity result in a variable
  int moisture= analogRead(A1);

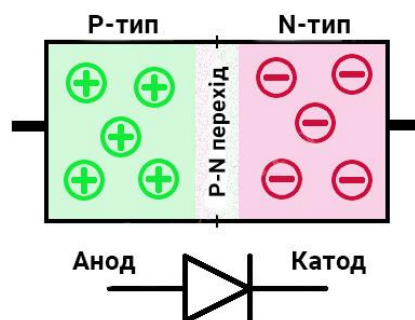
  Serial.println(moisture);
}
```

In general, a light sensor outputs binary data such as "0" when there are no light rays or "1" when there are light rays.

There is also an analogue of the lighting sensor - a photoresistor. In terms of properties, it is also similar, but it does not refer to sensors, but to resistors, such as: potentiometer; resistor; etc. Resistors are used to change the current, and sensors are used to change the voltage. This is how the photoresistor is marked on the diagrams:



There is also an analogue of a photoresistor and a light sensor - a photodiode, which belongs to diodes. A diode is an electronic component that passes electric current only in one direction - from the anode to the cathode. A diode is also called a rectifier, as it converts alternating current into pulsating direct current. A semiconductor diode consists of a plate of semiconductor material (silicon or germanium), one side of the plate has p-type electrical conductivity, that is, it accepts electrons. The other side donates electrons and has n-type conductivity. Contact metal layers are applied to the outer surface of the plate, to which the wire terminals of the diode electrodes are soldered.



This is how the photodiode is marked on the diagrams:



There are also resistors - a flexible sensor that changes the voltage when this resistor is deformed. His scheme:



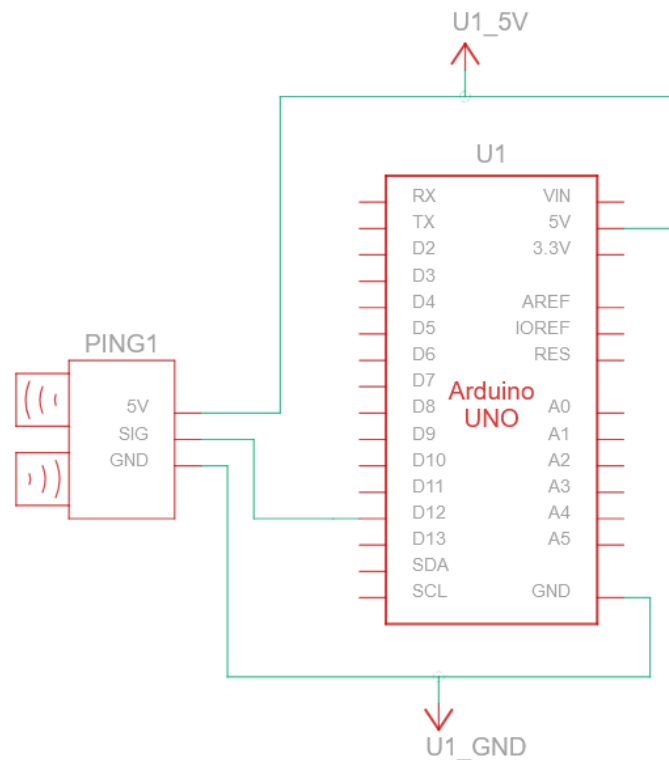
There is also a slope sensor made of resistors, which changes the voltage when the slope of the sensor changes. His scheme:



There is also an ultrasonic distance sensor, which belongs to the sensors. Ultrasonic distance sensor - a sensor that uses sound waves to determine the distance to an object. It recognizes an object from 200 centimeters and less. His scheme:



Now let's make a code for Arduino, which will show us the distance of some object from that sensor. Here is the scheme:



Now let's make the code for this scheme:

```
int sig = 0;

// This function should be done
// triggerPin - pin for sending a pulse
// echoPin - pin for echo reception
long readUltrasonicDistance(int triggerPin, int echoPin)
{
    pinMode(triggerPin, OUTPUT);
    digitalWrite(triggerPin, LOW);
    delayMicroseconds(2);
    digitalWrite(triggerPin, HIGH);
    delayMicroseconds(10);
    digitalWrite(triggerPin, LOW);
    pinMode(echoPin, INPUT);

    // Returns the result of the function
    return pulseIn(echoPin, HIGH);
}

void setup() // Continuation of the code on the next page
```

```

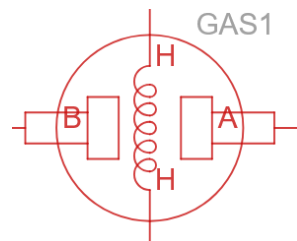
{
  Serial.begin(9600);
}

void loop()
{
  // Convert the distance in centimeters
  sig= 0.01723 * readUltrasonicDistance(12,12);

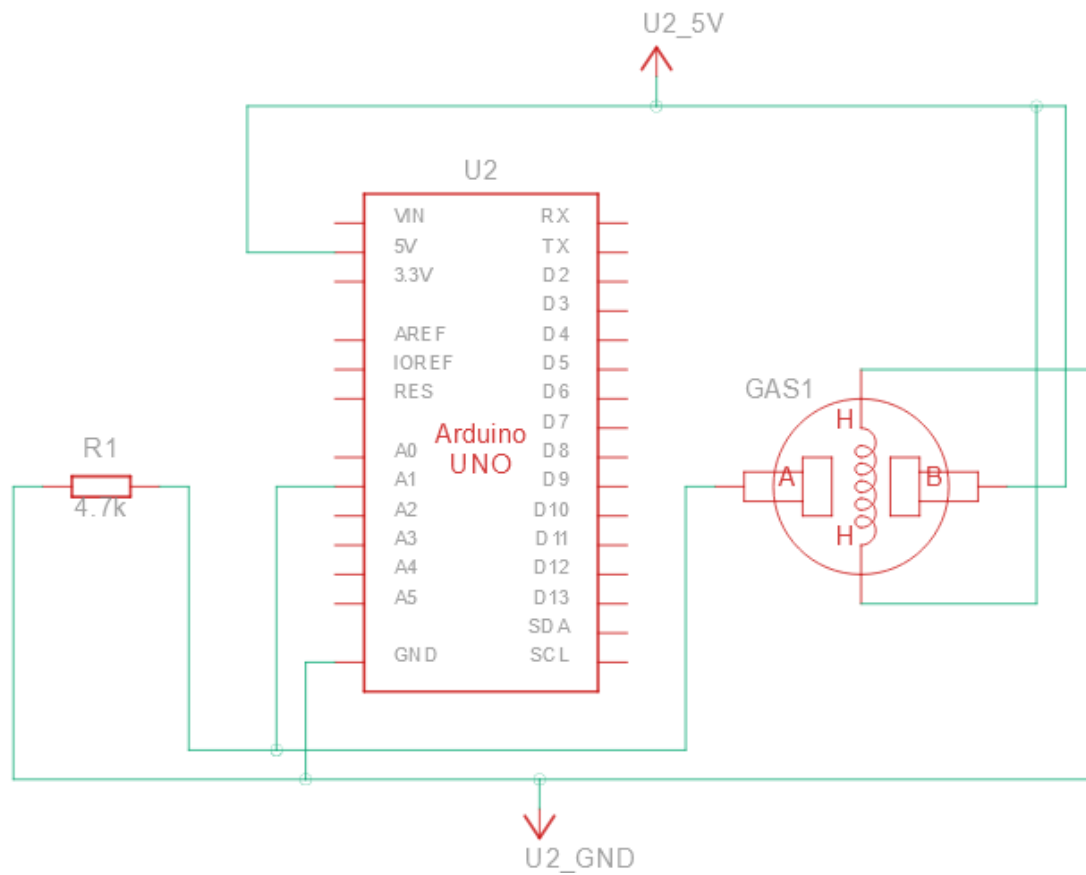
  // Display the result in the console
  Serial.println(sig);
}

```

There is also a gas sensor that detects gas such as: LPG (liquefied petroleum gases); isobutane; methane; alcohol; hydrogen; smoke; etc. This is how it looks on the diagrams:



Now let's make the code that will output data from the sensor. Scheme:



Code:

```
void setup()
{
  pinMode(A1, INPUT);
  Serial.begin(9600);
}

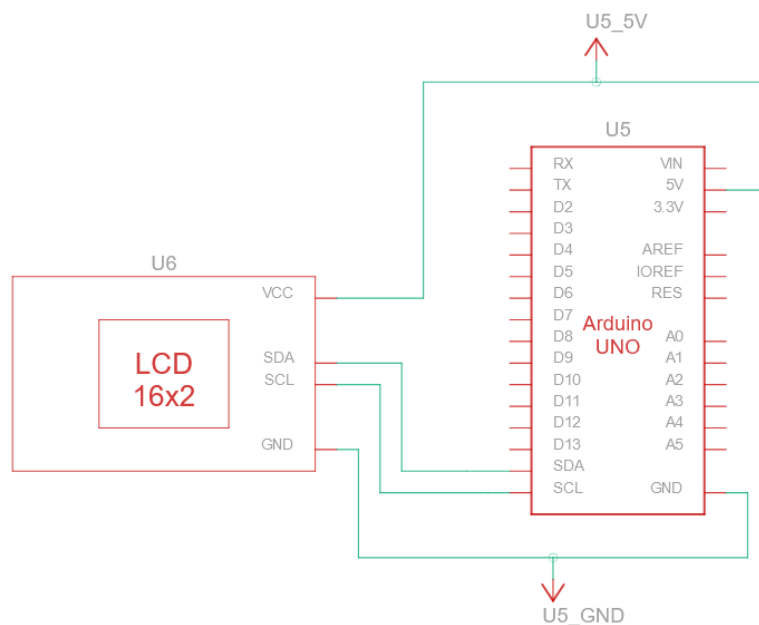
void loop()
{
  int moisture= analogRead(A1);
  Serial.println(moisture);
}
```


Lesson 2.5. USE OF LIQUID CRYSTAL SCREEN.

Liquid crystal screen - elements of radio electronics that display text on the screen. This is how this element is marked on the diagrams:



Now let's create the code for this scheme:



If we looked at elements that do not require library import, then in this case, when using a liquid crystal screen, the <Adafruit_LiquidCrystal.h> library should be imported.

Here is the circuit code:

```
#include <Adafruit_LiquidCrystal.h>

Adafruit_LiquidCrystal lcd_1(0);

void setup()
{
  lcd_1.begin(16,2);
}

void loop()
{
  // Continuation of the code on the next page
  lcd_1.setCursor(0,0); // Set coordinates for
  // display text
}
```

```

lcd_1.print("Hello World!"); // Display the text

lcd_1.setCursor(0,1); // Set coordinates for
// show the value of the variable
lcd_1.print(seconds); // Display the value of the variable

lcd_1.setBacklight(1); // Turn on the screen backlight
delay(500);
lcd_1.setBacklight(0); // Turn off the screen backlight
delay(500);
}

```

Lesson 2.6. USING A 7-SEGMENT SCREEN.

A 7-segment screen is a screen that consists of 7 segments for each digit. Here is how it is marked on the diagrams:



When assembling, the 7-segment screen is also connected in the same way as the liquid crystal screen. This can be seen in the diagram on page 60. In this way, you can also assemble radio equipment with a 7-segment screen.

```

#include "Adafruit_LEDBackpack.h"

Adafruit_7segment led_display1= Adafruit_7segment();

void setup()
{
  led_display1.begin(112);
}

void loop()
{
  led_display1.println("1234.");
  led_display1.writeDisplay();
}

```

Please note that only the following characters can be written on this screen: ".", ",", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "-", "_".

