

C++

Для радиоэлектронщиков

Вадим Савенко @ 2024

Содержание:

I РАЗДЕЛ. Знакомство с C++ (стр. 3 – 45)

- 1.1. ПОЧЕМУ СТОИТ ИЗУЧИТЬ C++ (стр. 3 – 7)
- 1.2. ПЕРЕМЕННЫЕ И ТИПЫ ДАННЫХ (стр. 7 – 17)
- 1.3. ОПЕРАТОРЫ (стр. 17 – 25)
- 1.4. РАЗОБРАЗОВАНИЕ (стр. 25 – 32)
- 1.5. ЦИКЛЫ (стр. 33 – 35)
- 1.6. ФУНКЦИИ (стр. 35 – 44)
- 1.7. ПРЕДОСТАВЛЕНИЕ СМОГИ ВВЕДЕНИЯ ДАННЫХ
ПОЛЬЗОВАТЕЛЕМ (стр. 45)
- 1.8. РАНДОМЫ (стр. 45)

II РАЗДЕЛ. ИСПОЛЬЗОВАНИЕ ARDUINO С ЯЗЫКОМ C++ (стр. 46 – 63)

- 2.1. ЧТО ТАКОЕ ARDUINO (стр. 46)
- 2.2. ИСПОЛЬЗОВАНИЕ РАДИОЭЛЕМЕНТОВ ИЗ ARDUINO (стр. 46 –
51)
- 2.3. ЧТО НАДО ЗНАТЬ ПРИ НАПИСАНИИ КОДА ДЛЯ ARDUINO (стр.
51 – 53)
- 2.4. ИСПОЛЬЗОВАНИЕ ДАТЧИКОВ (стр. 53 – 61)
- 2.5. ИСПОЛЬЗОВАНИЕ ЖИДКО-КРИСТАЛИЧЕСКОГО ЭКРАНА (стр.
61 – 63)
- 2.6. ИСПОЛЬЗОВАНИЕ 7-СЕГМЕНТНОГО ЭКРАНА (стр. 63)

I РАЗДЕЛ. Знакомство с C++.

Урок 1.1. ПОЧЕМУ СТОИТ ИЗУЧАТЬ C++.

Большой спрос

По сей день в разных отраслях промышленности C++ является лучшим выбором. Ее широкое использование в существующих системах и адаптивность новых вызовов продолжают способствовать ее популярности в различных отраслях.

Преимущества и Сила C++:

- **Высокая Производительность:** C++ известен своей высокой производительностью и эффективным использованием системных ресурсов.
- **Богатая Стандартная Библиотека:** имеет большую стандартную библиотеку, которая предоставляет широкий спектр структур данных, алгоритмов и утилит, что экономит время и усилия разработчиков.
- **Прочный Фундамент:** дает глубокое понимание компьютерных наук, что делает его отличной отправной точкой в мир программирования.
- **Навыки:** многие концепции, которые вы изучаете в C++, применимы к другим языкам программирования, что облегчает изучение дополнительных языков в будущем.
- **Поддержка Сообщества:** имеет большое и активное сообщество, что означает наличие большого количества ресурсов, библиотек и инструментов для разработчиков.

C++ – это язык программирования общего назначения, который поддерживает как низкоуровневую, так и высокоуровневую парадигму программирования, что делает ее действительно адаптивным и мощным инструментом.

- **Низкий уровень:** прямой контроль над аппаратным обеспечением, требующий глубокого понимания архитектуры компьютера, более сложный, но более точный.
- **Высокий уровень:** более удобное программирование, более быстрая разработка, легче обслуживать и имеет более читаемый для человека код.

Низкий уровень	Высокий уровень
<i>Преимущества</i>	
полный контроль при приготовлении сэндвича с нуля	быстро и удобно , поскольку вы заказываете сэндвич с оптимизированным приготовлением
<i>Недостатки</i>	
занимает много времени и требует навыков и опыта в приготовлении сэндвичей	ограниченная кастомизация , отсутствие прямого контроля над процессом приготовления сэндвичей

Каждая программа C++ должна иметь функцию `main()`. Синтаксис ее выглядит следующим образом:

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

- `int main()`: исходная точка программы. Она называется главной функцией, и именно с нее начинается выполнение программы.
- `{ }`: фигурные скобки определяют блок кода. Все, что находится внутри этих скобок, относится к главной функции и является частью логики программы.
- `return 0;`: обозначает конец программы и указывает, что она успешно завершилась. Значение 0 означает, что все хорошо прошло. Если были проблемы, это значение может быть иным в выводе.

Заметьте! Оператор `return 0;` не обязательно в конце главной функции. Если его не вставить, компилятор автоматически добавит его.

Стандартные библиотеки служат хранилищами заранее написанного, многократного кода, упрощающего выполнение типовых задач. Они экономят разработчикам время и усилия, предлагая хорошо проверенные, стандартизированные инструменты для создания программного обеспечения.

Представьте, что вы используете библиотеку как здание из готовых больших блоков. Это гораздо быстрее и удобнее, чем производить маленькие блоки с нуля и использовать их.

Почему следует использовать стандартные библиотеки и файлы

Написание кода с помощью библиотеки похоже на написание книги с помощью словаря. Мы можем легко заменить фразу одним словом без потери основного смысла. К примеру:

Текст	Другой текст, в котором были заменены слова на фразы
<i>В биологическом сообществе, где организмы взаимодействуют с физическим окружением, все должно быть в таком состоянии, в котором разные элементы равны или в правильных пропорциях.</i>	<i>В экосистеме все должно быть в равновесии</i>

Директивы препроцессора

Чтобы добавить внешние файлы в программу, вам нужно использовать директивы препроцессора. Это команды, управляющие препроцессором, инструментом, преобразующим код перед компиляцией. Синтаксис большинства директив препроцессора таков:

#директива параметр

- **#**- символ, указывающий на то, что это директива препроцессирования.
- **директива**: конкретная директива препроцессирования;
- **параметр**: связанное значение или аргумент для настоящей директивы.

Команда, которая добавляет внешние файлы в вашу программу, называется `#include`.

#include <ИМЯ>

Заметьте! Стандартные файлы присоединяются с помощью угловых скобок `< >`, но вы также можете создавать собственные файлы и присоединять их к вашему проекту аналогично, используя двойные кавычки `" "`.

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

Как работает #include

Посмотрите на код ниже и попытайтесь запустить его.

Перед нами два файла: `main.cpp`; `header.h`.

Код main.cpp:

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

Код header.h:

```
}
```

Вы получите ошибку об отсутствии `}`. Это сделано специально, чтобы показать, как работает `#include`. Мы можем создать отдельный файл, содержащий только `}` символ и включить его в файл `main.cpp` с помощью директивы `#include`.

Традиционной первой программой на любом языке программирования часто является простая программа, выводящая сообщения на консоль.

Ввод и Вывод

Самый распространенный способ вывода информации в консоль — это использование библиотеки `iostream`. Она означает входящий (input)/исходящий (output) поток.

Как мы уже знаем, чтобы добавить библиотеку в нашу программу, мы должны использовать `#include`.

```
#include <iostream>
```

Вывод символов

Поток `cout` является частью библиотеки `iostream` и используется для стандартного вывода.

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

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

- `std::` означает, что идентификатор является частью стандартной библиотеки, организованной в пространстве имен `std`.
- `cout`: отвечает за вывод символов и используется для отправки вывода.
- `<<`: используется для ввода данных в стандартный поток вывода.

Комментарии

Комментарии используются для документации и не влияют на выполнение программы. Существует два типа:

- **Однострочные комментарии:** Начинаются с `//` и простираются до конца строки.
- **Многострочные комментарии:** Заключение между `/*` и `*/`.

Задание.Создайте приложение, которое выводит ваше сообщение.

Для решения этой задачи мы должны использовать библиотеку `iostream` и использовать `main()`. Также нужно использовать команду для отображения информации:

```
std::cout<< "Сообщение";
```

Тогда мы получаем код:

```
#include <iostream>

int main()
{
    // Output your message:
    std::cout<< "Сообщение";
}
```

При выполнении действий кода мы получаем результат в консоли:

```
Сообщение
```

Урок 1.2. ПЕРЕМЕННЫЕ И ТИПЫ ДАННЫХ.

В области языков программирования, переменные играют немаловажную роль в хранении и манипулировании данными. Процесс создания переменной включает в себя два необходимых шага: объявление и инициализацию.

- **Объявления:** процесс ввода переменной компилятора. Он предусматривает указание типа данных переменной и, по желанию, предоставление ему имени.

- **Инициализация:** процесс присвоения значения объявленной переменной. Это критически важно, чтобы избежать попадания непредсказуемых или мусорных значений в переменную.

```
#include <iostream>

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

Примечание! Вы можете указать исходное значение для переменной во время объявления.

Также при объявлении нескольких переменных, содержащих данные одного типа, удобно указывать тип данных только один раз. К примеру:

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

Конвенции именования

Выбор значимых и описательных имен для переменных важен для написания понятного и легкого для поддержки кода.

```
int n; // Плохое именование
```

ИЛИ

```
int kilKistStudentiv; // Хорошее название переменной
```

Названия переменных в C++ могут состоять из букв и цифр, но не могут начинаться с цифр, также не могут содержать пробелы или арифметические операторы. Также избегайте использования названий, совпадающих с зарезервированными ключевыми словами. Это может привести к компиляции ошибок.

Так именовать переменные нельзя как вот в этом коде:

```
int if = 25; // Использует зарезервированное слово
int email@number = 25; // Содержит специальный символ
int my age = 37; // Содержит пробел
```



```
int 121A = 121; // Начинается с цифры
```

При объявлении переменной необходимо указать, какой тип данных мы будем в ней хранить. Для удобства работы с памятью есть типы данных для каждой ситуации.

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

Числовые

Эти типы необходимы для хранения числовых значений и манипулирования числовыми данными. Мы можем разделить их на две группы:

- Типы данных для целых чисел (например, 12);
- Типы данных для чисел с плавающей запятой (например, 0.12).

Булевы

Тип данных `bool` представляет два логических значения: нуль интерпретируется как `false` и один интерпретируется как `true`.

Свет включен = `true` = 1 Свет выключен = `false` = 0

Char

Тип данных `char` используется для хранения отдельных символов, которые могут включать буквы, цифры, знаки препинания и специальные символы.

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

Целые числа

Тип данных `int` может сохранять значение в диапазоне от 2,147,483,648 до 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;
}
```

Это происходит потому, что когда вы объявляете переменную типа `int`, она выделяет ровно 4 байта памяти вашего компьютера. А числа выше 2147483647 (или ниже -2147483648) не помещаются в эти 4 байта. К счастью, другие доступные типы данных могут выделить вам большее (или меньшее) места для ваших нужд. Вот таблица:

Тип данных	Диапазон	Размер
short	-32,768 – 32,767	2 байта
int	-2,147,483,648 до 2,147,483,647	4 байта
long	-9,223,372,036,854,775,808 до 9,223,372,036,854,775,807	8 байт

Итак, вы можете использовать `long` для хранения больших чисел (например, популяция мира). Вы также можете использовать `short`, если вы уверены, что ваше число не выйдет за пределы диапазона от $-32,768$ до 32,767 (например, для хранения возраста пользователя). Использование `short` займет меньше места.

```
#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;
}
```

Заметьте. Будьте осторожны с типом данных, который вы выбираете. Если диапазон типа превышен, компилятор C++ не скажет вам об этом, и вы получите неожиданное значение, не зная, что что-то не так.

Числа с плавающей запятой

Вышеприведенные типы данных предназначены для хранения целых чисел. Если мы попытаемся присвоить 1.6 одному из них, вот что мы получим:

```
#include <iostream>

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

Тип `int` игнорирует десятичную часть числа. Такая же история с `short` и `long`. Для хранения чисел с плавающей запятой (десятичных) следует использовать тип данных `float` или `double`.

Тип данных	Точность	Размер
<code>float</code>	7 десятичных знаков	4 байта
<code>double</code>	15 десятичных знаков	8 байтов

Вот пример использования `double` для хранения 1.6.

```
#include <iostream>

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

Заметьте. Поскольку тип `float` имеет точность всего 7 цифр, число 123.456789 уже выходит за пределы его диапазона. Это может привести к неточным результатам (как показано ниже). Поэтому лучше использовать `double` по умолчанию, если вы не уверены, что точность `float` достаточна.

```
#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;
}
```

Очевидно, что вы можете использовать float или double для хранения целых чисел, поскольку это десятичные числа с десятичной частью, равной 0. Однако, как правило, если переменная сохраняет значения, которые могут быть только целыми числами (например, население или предпочтение), следует использовать short/int/long.

```
#include <iostream>

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

Вы также можете сохранять текст в переменных, кроме символов с помощью char. Для этого нужно

- Подключить файл string
- Использовать расширение области видимости std
- Объявить переменную типа string

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

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

Строочные переменные могут содержать числа (в виде текста):

Если вы попытаетесь добавить две переменные строки, вы получите конкатенацию (это работает без пробелов):

```
#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;
}
```

То же произойдет с числами – они не будут добавлены алгебраически.

Массив – это набор элементов одного типа. Чтобы создать массив, следует выполнить следующие действия:

- Определите тип данных для элементов, которые вы собираетесь хранить в массиве;
- Присвоить массиву имя;
- Указать количество элементов в массиве, поместив это количество в квадратные скобки после его имени. К примеру:

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

Компилятор выпустит ошибку, если размер не указан в статических массивах.

Для инициализации массива нужно указать все его элементы в фигурных скобках:

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

Чтобы получить нужный нам элемент из массива, мы можем сослаться на него с помощью индексов. Каждый элемент массива имеет свой индекс, равно как каждый дом в вашем городе имеет свой адрес.

Заметьте. Индекс начинается с индекса 0, а не с 1.

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

Длина массива, приведенного выше, равна 6. Если мы создадим массив длиной 5 с этими числами, он выдаст ошибку.

```
#include <iostream>

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

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

Предположим, что в массиве есть больше частей, чем вы указали при объявлении. В этом случае возникнет ошибка компиляции, поскольку компилятор выделяет фиксированное количество памяти при объявлении массива. Это все равно, что пытаться налить больше воды в полный стакан.

Если в массиве меньше элементов, чем вы указали при объявлении, то все неинициализированные элементы будут равны нулю или будут иметь мусорные значения (непредсказуемые или произвольные данные).

```
#include <iostream>

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

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

Вы можете представить массив как книгу, в которой каждая страница (элемент) пронумерована (индекс). Данные в массиве можно изменять, для этого следует обратиться к элементу за индексом *i*, например, задать ему новое значение:

```
#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;

    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;
}
```

Массив может быть элементом другого массива, например, объявим массив, элементами которого будут другие массивы. Чтобы объявить многомерный массив, вам понадобится еще одна пара квадратных скобок:

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

- Первая пара скобок – это основной массив;
- Вторая пара скобок говорит о том, что элементами основного массива будут малые массивы.

```
#include <iostream>

int main()
{
    //creating multidimensional array
    int myArray[4][3]={000,00,0},// первый element of main array
    {111,11,1},// second element of main array
    {222,22,2},// третий элемент main array
    {333,33,3};// четыре элемента основного уровня
};

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

Мы создали массив под названием myArray, содержащий четыре элемента, каждый из которых является массивом с тремя элементами.

Размер переменной – это объем памяти, зарезервированный компилятором. Компилятор резервирует определенное количество байт из памяти вашего компьютера, исходя из типа используемых вами данных. Вы можете использовать `sizeof()`, чтобы узнать размер переменной или типа данных в байтах. К примеру:

```
#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++ позволяет выбрать тип с точным размером бит, например `int8_t`, `uint8_t`, `int16_t`, `uint16_t` и т.д. Чтобы использовать эти типы данных, необходимо включить заголовочный файл `<cstdint>`.

```
#include <cstdint>
```

Кроме того, мы можем заставить компилятор определить тип переменной самостоятельно, используя ключевое слово `auto`.

```
#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++ также позволяет переименовывать существующие типы данных под себя. Для этого используется `typedef`:

```
#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;
}

```

При компиляции строка typedef сообщает компилятору, что MY_NEW_TYPE – это просто тип double.

Урок 1.3. ОПЕРАТОРЫ.

Оператор присвоения (=): в программировании используется для присвоения переменной значения. Синтаксис выглядит следующим образом:

```

#include<iostream>

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

    yourVar = myVar; //assign yourVar значение myVar

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

```

С типом данных string это работает точно так же:

```

#include <iostream>
#include <string>

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

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

```

Операторы равенство (==) и неравенство (!=) используются для численного сравнения 2 переменных:

```
#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;}

```

Почему 1 и 0? Это альтернативный подход к использованию булевого типа данных. Когда выражение `var == 9` является true, оно представляется как 1, и это означает, что `var` действительно равно числу 9. И наоборот, когда выражение `var == -9` имеет значение false, оно представляется как 0, что означает, что `var` не равно числу -9.

Оператор неравенство (!=) действует с точностью до наоборот:

```
#include<iostream>

int main()
{
    int var = 9;

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

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

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

```

Эти пять математических операторов (+, -, *, / и %) служат для выполнения разных математических операций:

```
#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;
    // Продолжение кода на следующей странице
    //using the multiplication operator(*)
    int resultMult = myVar * yourVar;
    std::cout<< "9 * 5 = " << resultMult << std::endl;

    //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;
}
```

Оператор остатка от деления (modulo)(%) вычисляет и возвращает остаток стандартной операции деления.

Оператор деления (/) возвращает только часть результата, отбрасывая остаток. К примеру, при делении 10 на 3 результатом будет 3, а не 3.333... Чтобы получить нужный результат деления с десятичными знаками (например, $10/3 = 3.333$), необходимо, чтобы хотя бы один из операндов имел тип данных double или float.

```

#include<iostream>

int main()
{
    // один из variable должен быть двух или float типа
    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;
}

```

Оператор сумма является единственным математическим оператором, который применим к строке (он называется конкатенация):

```

#include<iostream>
#include<string>

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

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

```

Есть также операторы сравнения (>, <, <=, >=). Они используются, когда вам нужно численно сравнить значение с определенным диапазоном:

```
#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;
}
```

Для одновременного вычисления нескольких условий вы должны использовать логические операторы НЕ (!), ТА (&&) и ИЛИ (||). Давайте проиллюстрируем использование логических операторов в реальных сценариях:

- Я возьму кошелек на прогулку, если на моем пути есть банк И магазин;
- Я не возьму кошелек на прогулку, если по дороге будет не банк;
- Я возьму кошелек на прогулку, если по дороге есть банк ИЛИ магазин.

```

#include <iostream>

int main()
{
    bool bank = true;
    bool shop = true;

    //using AND (&&) operator
    bool wallet1 =(bank && shop);
    std::cout<< "Это является банком и магазином, wallet = " << wallet1 << std::endl;

    //using NOT(!) operator
    bool wallet0 =(!bank);
    std::cout<< "There is not bank, wallet = " << wallet0 << std::endl;

    //using OR (||) operator
    bool wallet2 =(bank || shop);
    std::cout<< "Это является банком или wallet = " << wallet2 << std::endl;
}

```

Составленный оператор присваивания- это комбинированный оператор, сочетающий в себе оператор присваивания и арифметический (или побитный) оператор. Результат такой операции присваивается левому операнду (находящемуся перед знаком =).

```

#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;
}

```

Использование другого составного оператора присвоения:

```
#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;

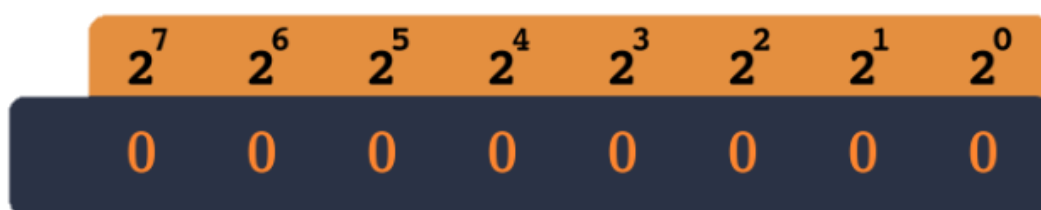
    value *= 4; // value = value * 4
    std::cout << "value = 16 * 4 = " << value << std::endl;

    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;
}
```

Что такое байт

Это б, базовая единица хранения цифровой информации. Байт состоит из 8 битов.



В основе хранения цифровой информации лежит байт, состоящий из 8 битов. Каждый бит представляет собой цифру, а числа обычно выражаются в двоичной системе счисления. Двоичное представление предполагает выражение числа как суммы степеней 2.

Примеры двоичного представления

1010=10

111111=255

101010=42

Рассмотрим число 4, которое в двоичной системе счисления имеет вид 00000100. Каждый бит в этом байте соответствует степени 2:

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

Аналогично, число 5 представляется как 00000101:

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

Bitwise shift

Поразрядное смещение

- В этом контексте один сдвиг влево: значение умножает значение на 2.
- И напротив, когда мы выполняем один битовый сдвиг вправо: значение делится на 2.

```
#include <iostream>

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

Считайте операцию сдвига вправо и влево >> ИЛИ << на n позиций как деление или умножение на 2^n .

В случае $10 \gg 3$ это фактически деление 10 на 2^3 (что равно 8), а результат равен 1.

Математические операторы, необходимые для следующей задачи

Оператор	Описание
-	Вычитает одно значение от другого
*	Умножает два значения
/	Делите одно значение на другое
+	Добавляет одно значение к другому
%	Показывает остаток при делении одного значения на другое
**	Возвести к степени
//	Деление нацело

Урок 1.4. Разветвления.

Конструкция if...else в программировании позволяет вашей программе выбирать разные пути и управлять разными потенциальными результатами.

Она состоит из двух основных компонентов: условия и ответных действий или последствий на основе этого условия.

Вот иллюстрация:

```
#include<iostream>

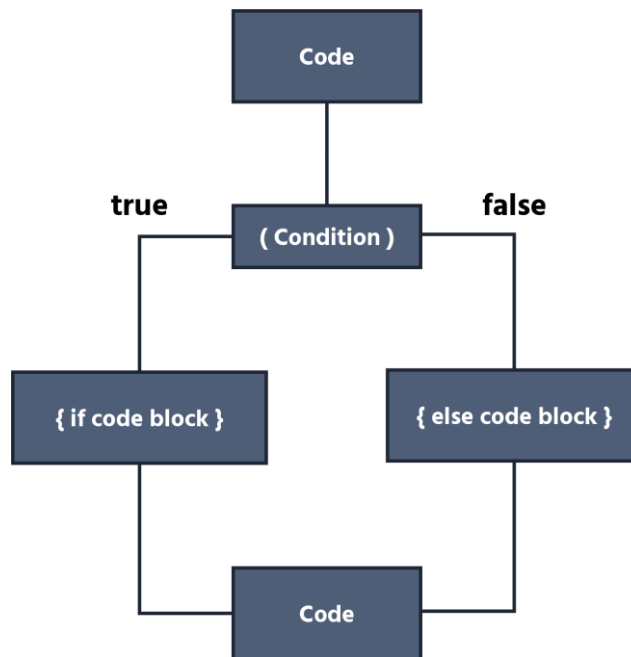
int main()
{
    int var = 13;

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

    if(var == 13)
    {
        std::cout<< "13 == 13, это 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;
    }
}
```



Существует также обработка противоположного случая с помощью else:

```
#include<iostream>

int main()
{
    int var = 200;

    /* 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;
    }
}
```

Внутри if...else могут быть другие 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;
        }
    }
}
```

Вы также можете использовать конструкцию else if:

```
#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;
    }
}
```

Оператор терминальный оператор предлагает лаконичную альтернативу оператору `if...else`, с заметным отличием. Он состоит из трех ключевых элементов:

1. Булевоe выражение;
2. Инструкции для случая `истина`;
3. Инструкции для случая `false`.

```
(булевыe выражение)?инструкция_для_истинного_случая:инструкция_для_ложного_случая
```

Также можно делать следующим образом для получения результата в консоли:

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

int main()
{ //Продолжение кода на следующей странице
```

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

Такой оператор удобно использовать, например, при сравнении двух чисел:

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

В этом случае результат тройной операции был присвоен переменному результату.

Когда сравнение возвращает истинный результат, значение var1 будет сохранено в переменном result.

if true then

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



И наоборот, если результат сравнения false, переменной result будет присвоено значение переменной var2.

if false then

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



Заметьте.Соблюдайте совместимость типов данных!

Как бы это выглядело с использованием 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;
}
```

Конструкция switch-case позволяет сравнивать результат выражения с набором заранее определенных значений. Структура switch-case:

```
switch(someExpression)
{
    case someCheck1:
        // если этот случай разрешен
        сделать_что-то1;
        break;

    case someCheck2:
        // если этот случай разрешен
        сделать_что-то2;
        break;

    по умолчанию:
        //если ни один из случаев не подходит
        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- это оператор, означающий выход из блока кода;
- default– это необязательная, но полезная часть. Эта часть будет выполнена, если ни один из вариантов не подойдет.

В нашем случае мы проверяем переменную, если она равна 5, тогда будет отображен соответствующий текст, и, используя оператор break, поток

программы выйдет из всей конструкции `switch-case`, и не будет обработки других случаев.

- `break`- оператор означает выход из блока кода.
- `default`– необязательная, но полезная часть. Эта часть будет выполнена, если ни один из случаев не подходит.

В нашем случае мы проверяем переменную, если она равна 5, то выводится соответствующий текст и с помощью оператора `break` поток программы покинет всю конструкцию `switch-case`, и не будет никакой обработки других случаев.

```
#include <iostream>

int main()
{
    int variable = 5;

    switch(variable)
    {
        case 5:
            std::cout<< "Value of variable equals 5" << std::endl;
            // delete "break;"

        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;"
    }
}
```

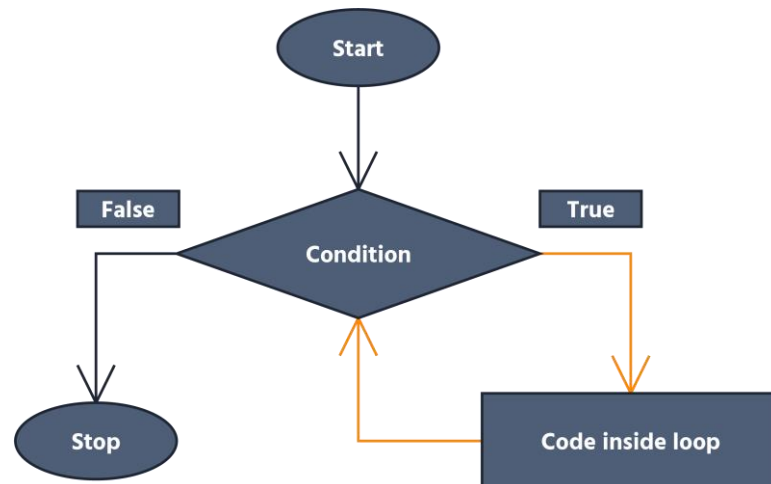
Но оператор "переключения" имеет одну оговорку. Мы намеренно изымаем оператор `break`:

Мы использовали `if...else`, `switch-case` для сравнения наших переменных с другими значениями. Но что если нам нужно сделать что-то сто раз? Тысячу раз? Миллион раз?

Именно для таких случаев и предназначены циклы! Они позволяют заиклнить вашу программу при определенных условиях. Структура цикла `while`:

Урок 1.5. ЦИКЛЫ.

```
while(someExpression== true)
{
// если someExpression == true, то 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;
}
```

В этом случае мы подсчитали ($x+=1$) 992 раза. Цикл выполнялся до тех пор пока $x+u$ не стало равным результату (1000).

Как только выражение $x+u$ стало равным результату, цикл завершился, и мы получили корень уравнения (x).

Заметьте. Если условия не выполняются, цикл может не начаться.

Важно убедиться, что цикл имеет условие выхода, то есть цикл не будет бесконечным. Пример нескончаемого цикла:

```
#include <iostream>

int main()
{
    bool condition = true;

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

В отличие от цикла `while`, который может никогда не выполниться, цикл `do...while` гарантированно выполняется по крайней мере один раз.

Структура цикла `do...while`:

```
do
{
    //сделать_что-то;
}
while(someExpression== true);
```

Заметьте. Строка, содержащая часть `while`, заканчивается точкой с запятой (;).

Цикл `for` сложнее других циклов и состоит из трех частей. Структура цикла `for`:

```
for(счетчик; условие выхода; выражение цикла)
{
    // блок инструкции
    сделать_что-то;
}
```

- Счетчик;
- Условие выхода;
- Выражение цикла.

```
#include <iostream>

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

- `int counter = 0`: iteration counter;
- `counter++`: Для каждой итерации, 1 will be added to the `countervariable` to mark the passage of the loop;
- `counter <= 5`: loop termination condition. The loop will continue if the `countervariable` является неправильным или эквивалентным 5.

Теперь вопрос, Сколько итераций сделает этот цикл?

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

А) 5; Б) 6; в) 1; Г) 0;

Урок 1.6. ФУНКЦИИ.

Функции – это небольшие подпрограммы, которые можно вызвать при необходимости. Каждая функция имеет имя, по которому ее можно вызвать.

```
int main()// "main" – это имя функции
{
    return 0
}
```

Заметьте. Имя `main` уже зарезервировано на языке C++. Поэтому при объявлении функции с таким именем компилятор выпустит ошибку.

Чтобы создать функцию, нужно:

- определить тип данных, которые она будет возвращать;
- присвоить ей имя;
- предоставить блок инструкций (тело) в фигурных скобках { . . . } для определения его функциональности.

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

Например, создадим функцию, выводющую текст "c<>definity":

```
std::string nameOfCourses()// тип и имя функции
{
    // начало тела

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

}
// конец тела
```

```
#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;
}
```

Создадим функцию, упрощающую процесс превращения температур с Фаренгейта в Цельсия. Это практическое применение в повседневной жизни.

```
#include <iostream>

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

```

    return celsius;
} //продолжение кода на следующей странице

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

```

Примечание! Аргумент функции представлен переменной degree, содержащей данные, с которыми функция работает. В этом контексте она относится к температурам в градусах Фаренгейта, которые следует превратить в градусы Цельсия. Позже мы более подробно рассмотрим аргументы функций.

Компилятор обрабатывает наш программный код последовательно, подобно тому, как человек читает книгу, и если он встречает неизвестные имена переменных или функций, он выведет ошибку.

Например, попробуем вызвать функцию до ее определения.

Этот пример вызывает ошибку. Это сделано намеренно.

```

#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;
    returncelsius;
}

```

В таких ситуациях важно использовать прототипы функций.

Цель прототипа состоит в том, чтобы заранее сообщить компилятору о нашей функции. Создание прототипа похоже на стандартное объявление функции, но с тонкой разницей:

- указать тип будущей функции;

- дать ей имя;
- **аргументы**(при необходимости);
- поставить знак конца выражения.

```
int FahrenheitToCelsius(int degree); //прототип нашей функции
```

Добавим прототип функции к нашему примеру, вызвавшему ошибку:

```
int FahrenheitToCelsius(int degree); //прототип нашей функции
```

Примечание!Прототипирование является полезным при работе с большим количеством функционала. Чтобы избежать "мусора" в основном файле, прототипы и определения функций переносятся в сторонние файлы и подключаются к основному файлу с помощью директивы `#include`.

```
#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;
}
```

Примечание!Прототипирование является полезным при работе с большим количеством функций. Чтобы избежать "мусора" в главном файле, прототипы и определение функций перенесены в сторонние файлы и включены в основной файл с помощью директивы `#include`.

При создании функции всегда указывается тип возвращаемых данных.

В примере функции `main` мы видим, что она имеет целочисленный тип данных, а это значит, что по завершении работы она вернет целое значение, в нашем случае число 0.

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

Заметьте.Поскольку функция `main` зарезервирована в C++, она всегда будет возвращать целое число.

Но наши функции могут возвращать любое значение:

```
double notMainFunc()
{
    return 3.14;
}
```

Чтобы вызвать функцию, нужно написать ее имя в скобках:

```
notMainFunc();
```

```
#include <iostream>

double notMainFunc()
{
    return 3.14;
}

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

Мы создали функцию, которая возвращает значение 3.14 в качестве `double` тип данных, и мы вызвали эту функцию, чтобы показать ее вывод на экране.

Функции также могут быть типа `string`:

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

```

std::string notMainFunc();//string function
{
    return "codefinity";
} //продолжение кода на следующей странице

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

```

typedef также можно применять:

```

#include <iostream>

typedef int MY_NEW_TYPE;

MY_NEW_TYPE TYPEfunc();//MY_NEW_TYPE функция
{
    return 777;//продолжение кода на следующей странице
}

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

```

Если вы не можете точно указать тип возврата, оператор auto заставит компилятор сделать это за вас:

```

#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;
}
```

Оператор return завершает выполнение функции и возвращает значение определенного типа.

```
int func()//int – заранее определен
{
    переменнаяint = 10;
    вернутьпеременную;//переменная = 10
}
```

Если тип неправильно указан, функция будет вести себя непредсказуемо:

```
#include <iostream>

unsigned short func()
{
    return -10;
}

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

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

То есть, перед созданием функции необходимо указать тип возвращаемых данных. Также в C++ есть особые функции void. Функции с таким типом данных могут не возвращать ничего или вернуть ничего.

```
#include <iostream>

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

    //function without return
}

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

```
//продолжение кода на следующей странице
```

```
#include <iostream>

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

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

Заметьте.Обычно функции типа void просто выводят статический текст или работают с указателями

```
#include <iostream>

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

    return a;
    return b;
}

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

Функция с параметром (аргументом) – это функция, которая должна работать с объектом "извне".

```
int func(int argument)
{
    вернуть аргумент;
}
```

Примечание. Аргумент функции – это локальная переменная, которая создается и существует только в текущей функции.

К примеру, давайте создадим функцию, которая делит любое целое число на 2:

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

Чтобы использовать такую функцию, нужно вызвать ее и передать ей аргумент:

```
func(10);
```

Приведем пример:

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

В функцию можно передавать несколько аргументов:

К примеру, создадим функцию, которая делит любое целое число на 2:

`func(5, 7)` – вызов функции и передача ей аргументов. Аргументы передаются через запятую (,).

Также можно передавать массивы:

Чтобы использовать такую функцию, вы должны вызвать ее и передать аргумент.

```
func(10);
```

Вот пример:

```
#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);
}

```

Вы можете передавать несколько аргументов в функцию:

```

#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) – вызов функции и передача аргументов в нее. Аргументы передаются через запятые (.). Вы также можете передавать массивы:

```

#include <iostream>

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

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

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

```

Урок 1.7. ПРЕДОСТАВЛЕНИЕ СМОГИ ВВЕДЕНИЯ ДАННЫХ ПОЛЬЗОВАТЕЛЕМ.

Если мы знаем о `std::cout` для отображения текста, но можно спрашивать пользователя, чтобы он написал данные для переменной с помощью команды `std::cin`. Такой код выглядит следующим образом:

```
#include <iostream>

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

Урок 1.8. РАНДОМЫ.

Для нахождения случайного числа, выдаваемого компьютером, нужно сделать randomness чисел в коде. Такой код выглядит следующим образом:

```
#include <iostream>

int main()
{
    int n=/*От какого числа будет рандом*/;
    int m=/*К какому числу будет рандом*/;

    /*Нахождение рандомного числа*/
    int random=n+ rand()%m;

    /*Показать рандомное число в консоли*/
    std::cout<<random<< std::endl;
}
```

II РАЗДЕЛ. ИСПОЛЬЗОВАНИЕ ARDUINO С ЯЗЫКОМ C++.

Урок 2.1. ЧТО ТАКОЕ ARDUINO.

Arduino (Ардуино) — аппаратная вычислительная платформа для любительского конструирования, основными компонентами которой является плата микроконтроллера с элементами ввода/вывода и среда разработки Processing/Wiring на языке программирования, упрощенного подмножеством C/C++. Arduino может использоваться как для создания автономных интерактивных объектов, так и подключаться к программному обеспечению, выполняемому на компьютере (например: Processing, Adobe Flash, Max/MSP, Pure Data, SuperCollider). Информация о плате (рисунок печатной платы, спецификации элементов, программное обеспечение) находятся в открытом доступе и могут использоваться теми, кто предпочитает создавать платы собственноручно.

Название Arduino произошло от бара в Ивреа, Италия, где встречались некоторые из основателей проекта. Бар был назван в честь Ардуина I, который был маркграфом Марша Ивреи и королем Италии с 1002 по 1014 годы.

Урок 2.2. ИСПОЛЬЗОВАНИЕ РАДИОЭЛЕМЕНТОВ С ARDUINO.

Для изучения использования радиоэлементов с Arduino нужно сначала узнать об использовании радиоэлементов без Arduino.

Радиоэлектроника – отрасль науки и техники, охватывающая теорию, методы создания и использования устройств для передачи, приема и преобразования информации с помощью электромагнитной энергии.

Термин «радиоэлектроника» появился в 50-х годах 20 века и является в известной степени условным. Радиоэлектроника охватывает радиотехнику и электронику, в том числе полупроводниковую электронику, микроэлектронику, квантовую электронику, ИК технику, хемотроник, оптоэлектронику, акустоэлектронику, криоэлектронику и другие радиоэлектроники тесно связана, с одной стороны, с радиофизикой, физикой, твердым телом, а с другой – с электротехникой, автоматикой, телемеханикой и вычислительной техникой.

Методы и средства радиоэлектроники находят широкое применение в радиосвязи, космической технике, системах дистанционного управления,

радионавигации, автоматике, вычислительной технике, радиолокации, военной и специальной технике, в бытовой технике и т.д. Производцией радиоэлектроники есть радиоэлектронная аппаратура.

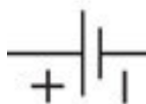
Сфера использования радиоэлектроники непрерывно расширяется, проникая в экономику, промышленное производство, сельское хозяйство, медицину, транспорт и другие области человеческой деятельности.

Для подачи сигнала в электронике необходим источник питания. В общем источником питания могут быть: батарея; аккумулятор; вилка с проводом; тому подобное. Каждое питание имеет заряды, такие как «Анод» и «Катод». Катод – электрод некоторого прибора, подсоединенный к отрицательному полюсу источника тока, а анод – электрод, от которого в газе или электролите направлен ток (движение положительных зарядов).

В литературе встречаются разные обозначения знака катода - "-" или "+", что определяется, в частности, особенностями описываемых процессов. В электрохимии принято считать, что «-» катод – электрод, на котором происходит процесс восстановления, а «+» анод – тот, где проходит процесс окисления. При работе электролизера (например, при рафинировании меди) внешний источник тока обеспечивает на одном из электродов избыток электронов (отрицательный заряд), здесь происходит восстановление металла, это катод. На другом электроде обеспечивается нехватка электронов и окисление металла, это анод. Вместе с тем, при работе гальванического элемента (например, медно-цинкового), избыток электронов (и отрицательный заряд) на одном из электродов обеспечивается не внешним источником тока, а собственно реакцией окисления металла (растворение цинка), то есть в гальваническом элементе отрицательным, если следовать приведенному определению, будет анод. Электроны, проходя через внешний круг, тратятся на прохождение реакции восстановления (меди), то есть катодом будет положительный электрод. Так, на приведенной иллюстрации изображен обозначенный знаком «+» катод гальванического элемента, на котором происходит восстановление меди. Согласно такому толкованию, для аккумулятора знак анода и катода изменяется в зависимости от направления протекания тока.

В электротехнике направлением тока принято считать направление движения положительных зарядов, поэтому в вакуумных и полупроводниковых приборах и электролизных ячейках ток течет от положительного анода к отрицательному катоду, а электроны, соответственно, наоборот, от катода к аноду.

В общем о разработке радиотехники, постоянно делается схемы устройства. Вот так обозначают источник питания:




В электронике элементы разделяются для ввода информации (такие как; резисторы; диоды; датчики; и т.п.) и для вывода информации (такие как: светодиоды; бужер; двигатель; и т.п.). Светодиод – элемент, работающий как лампочка. Бужер – элемент, производящий звуковой шум.

Вот таблица, как обозначаются элементы:

Элемент	Схематическое изображение
Источник питания	
Светодиод	
Бужер	
Динамик	
Кнопка	
Постоянный резистор	или
Сменный резистор (Потенциометр)	
Резистор с указанием мощности	
Диод	A K

Стабилитрон	
Конденсатор	
Полярный конденсатор	
Переключатель	
Двигатель	
Индуктор	
Вольтметр	
Амперметр	
Фоторезистор	
Реле	
NPN-транзистор	

PNP-транзистор	
----------------	--

Тактовую кнопку (иногда называют "пуш-батон") – это переключатель, который при нажатии запирает или размыкает электрическую цепь. Она имеет два состояния: замкнутое (при нажатии) и разомкнутое (когда не нажатое). Используется для запуска или остановки различных электронных устройств или выполнения определенных команд.

Резистор – это компонент, ограничивающий ток в электрической цепи. Он имеет определенное сопротивление, измеряемое в омах (Ω). Резисторы используются для контроля тока, распределения напряжения и защиты других компонентов избыточного тока.

Потенциометр – это сменный резистор, сопротивление которого можно регулировать вручную. Потенциометры часто используются как регуляторы громкости в аудиоаппаратуре или для настройки различных параметров в электронных схемах.

Фоторезистор (или LDR – Light Dependent Resistor) – это резистор, сопротивление которого изменяется в зависимости от интенсивности света. Он используется в датчиках освещения и в различных автоматических системах, где требуется реакция на изменения освещенности.

Диод – это полупроводниковый компонент, пропускающий ток только в одном направлении. Он используется для выпрямления тока, защиты схем от обратной полярности и для других целей в электронных устройствах.

Стабилитрон – это специальный вид диода, стабилизирующий напряжение на определенном уровне. Он используется в схемах для обеспечения стабильного питающего напряжения или защиты от перепадов напряжения.

Конденсатор – это компонент, сохраняющий электрический заряд. Он имеет две обкладки, разделенные изолятором (диэлектриком). Конденсаторы используются для фильтрации сигналов, уменьшения пульсаций, хранения энергии и других целей.

Полярный конденсатор – это тип конденсатора, у которого есть полярность: положительный и отрицательный выводы. Он обладает большей емкостью по сравнению с неполяризованными конденсаторами,

но требует правильной полярности подключения. Используется в схемах постоянного тока.

Амперметр – это измерительный прибор для измерения силы тока в электрической цепи. Его измерительные единицы – амперы (А).

Амперметр подключается последовательно к цепи.

Вольтметр – это измерительный прибор для измерения напряжения в электрической цепи. Напряжение измеряется в вольтах (В). Вольтметр подключается параллельно компонентам цепи.

NPN-транзистор – это тип биполярного транзистора, где два слоя полупроводникового материала типа "n" окружены одним слоем типа "p". Он часто используется в схемах для усиления или переключения сигналов.

PNP-транзистор – это биполярный транзистор, в котором два слоя полупроводникового материала типа "p" окружены слоем типа "n". Транзисторы PNP также используются для усиления и переключения сигналов, но работают в обратном режиме по сравнению с NPN-транзисторами.

Резистор с указанием мощности – это резистор, для которого указана максимальная мощность, которую он может выдерживать без перегрева и повреждения. Мощность измеряется в ваттах (Вт). Выбор резистора с соответствующей мощностью важен для предотвращения его перегрева и выхода из строя в схеме.

Для дальнейшего изучения радиоэлектроники советуем перечитать книги «Р. Токхейм: Основы цифровой электроники» и «П. Хоровиц, У. Хилл: Искусство схемотехники»

Урок 2.3. ЧТО НАДО ЗНАТЬ ПРИ НАПИСАНИИ КОДА ДЛЯ ARDUINO.

В общем, при написании кода для Arduino мы используем файл «sketch.ino», который имеет такой код:

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

Функция `setup()` выполняет все команды, которые у себя имеет, при запуске Arduino (В общем это могут быть команды для настройки элементов, такие как: жидкокристаллический экран; серво; двигатель постоянного тока; и т.п.), а функция `loop()` Arduino выполняет постоянно все команды, которые у себя есть.

Также следует помнить о командах, которые используются для Arduino. Вот код с базовыми командами в функции `loop()` для постоянного выполнения команд:

```
void loop() {
  /*Вывеститекстовое значение в консоли.
  Можно вывести буквы английского языка*/
  Serial.println("Hello World!");

  //Таймер для ожидания времени в миллисекундах
  delay(1000); // Ожидать 1 секунду

  //Назначить вывод "13" на "ВЫСОКИЙ"
  digitalWrite(13, HIGH);

  //Назначить вывод "13" на "НИЗКИЙ"
  digitalWrite(13, LOW);
  /*Выводы в Arduino:

  Аналоговые пены: A0; A1; A2; A3; A4; A5;
  Цифровые пены: 0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13;
  Питание: 3.3V; 5V;
  Заземление: GND;

  Аналоговая непрерывная связь (например: свет; звук), а цифровые
  используют для чтения и передачи цифрового сигнала, такие как "0"
  или "1". */

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

  // Условие, если оба оператора сравнения равны "True"
  if(a==b&&c==d) {
    ...
  }else{
    ...
  }
}
```

```

// Условие, если хотя бы оператор сравнения равен "True"
if(a==b&& c==d) {
    ...
} else{
    ...
}
}

```

При создании схемы с использованием Arduino, она на схеме обозначается следующим образом:

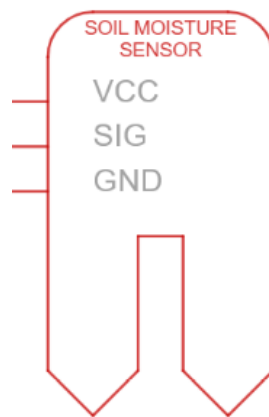


При создании схемы с использованием Arduino, она на схеме обозначается следующим образом:

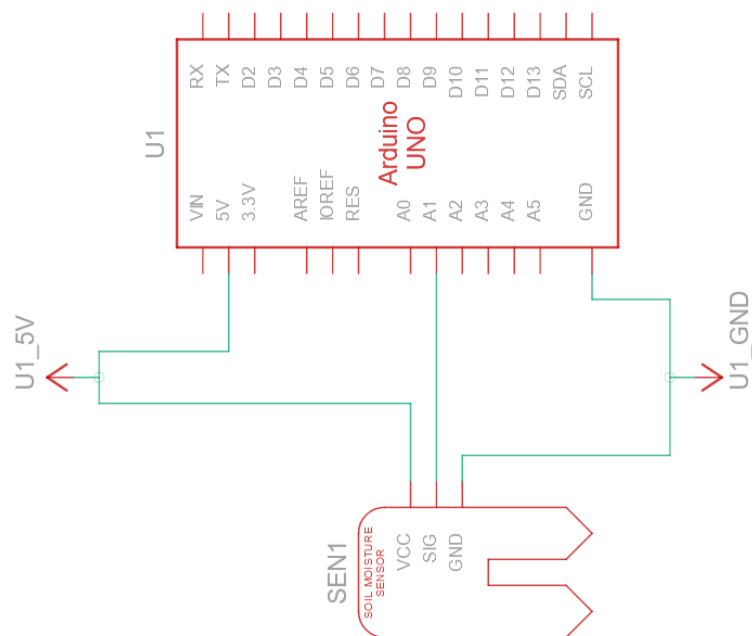
Урок 2.4. ИСПОЛЬЗОВАНИЕ ДАТЧИКОВ.

Датчики – элементы радиоэлектроники, принимающие сигнал от чего-либо. Например: датчик газа принимает сигнал благодаря газу; датчик влажности принимает сигнал благодаря воде и влажности; фоторезистор (датчик наличия света) принимает сигнал от освещения окружающего мира; тому подобное. В целом датчики имеют 3 вывода: питание (VCC); заземление (GND); сигнал (SIG). Есть также резисторы, так как: постоянные резисторы; потенциометры; фоторезисторы; и т.д., которые требуют только питания и имеют 2 выхода, где от одного идет питание, а из другого выходит питание и получаем сигнал.

Для начала разберемся с датчиком влажности. Датчик влажности служит для получения сигнала от воды и пространства влажности. Вот так он сказывается в схемах:



Давайте сделаем код Arduino, который будет отправлять в консоль информацию о влажности. Вот его схема:



А вот его код:

```
int moisture = 0;

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

void loop()
{
  moisture = analogRead(A1); // Получаем результат
  // влажности в переменной
  Serial.println(moisture);
}
```

Теперь перейдем к датчику температуры. Термистор (датчик температуры) служит для получения информации о температуре в окружающей среде. При нагревании окружающей среды увеличивается напряжение термистора. Есть много разновидностей датчиков температуры, такие как: TMP36; LM35; тому подобное. Но лучшим датчиком для определения температуры – TMP36, потому что он может определять как температуру меньше нуля, так и температуру больше нуля, потому что он получает данные размера аналогового значения. Формулы для нахождения градусов Цельсия и напряжения от аналогового входа TMP36 есть в начале следующей страницы.

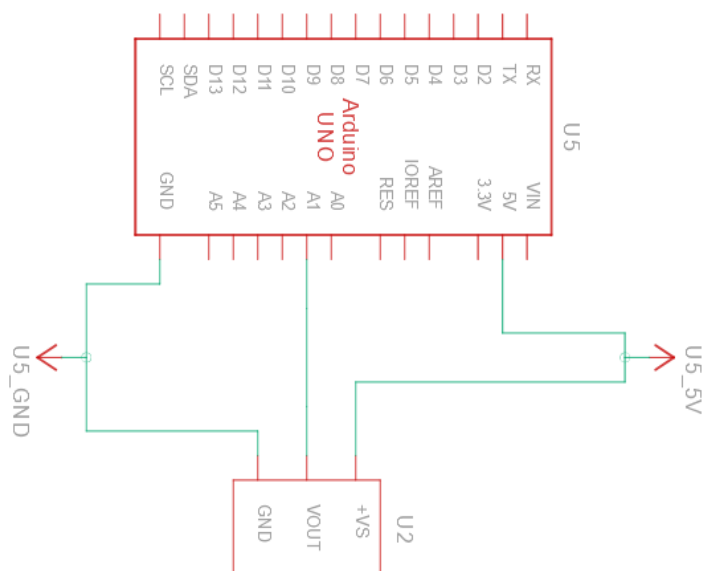
$$\text{Напряга (Вольт)} = \frac{(\text{Аналоговое значение}) * 1.1}{1024}$$

$$\text{Температура (Цельсиях)} = (\text{Напряга} - 0.5) * 100$$

Если TMP36 получил сигнал в 50 градусов по Цельсиям, то при выходе он даст 1 вольт. Вот как сказывается TMP36 на схеме:



Теперь сделаем код, который будет в консоли выдавать информацию о вольте и температуре в градусах Цельсия от термистора. Вот схема:



Вот код схемы:

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

void loop()
{
  // Продолжение кода на следующей странице
  int reading= analogRead(A1);
  float voltage=(reading* 1.1)/ 1024.0;
  float temperatureC=(voltage- 0.5)* 100;

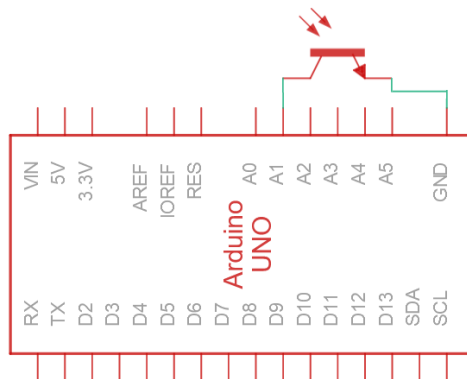
  // Первая строка
  Serial.print(voltage);
  Serial.println("volts");

  // Вторая строка
  Serial.print(temperatureC);
  Serial.println(" degrees C");
}
```

Есть датчики освещения, которые служат для выдачи сигнала об освещении пространства. Датчик освещенности – это полупроводниковый элемент, сопротивление которого изменяется при изменении световых лучей. При наличии освещения он пропускает напряжение. Вот как он сказывается в схемах:



Теперь сделаем код, при котором датчик освещения если есть наличие световых лучей, получаем в консоли «1», а в другом случае у «0». Вот схема:



Вот код:

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

// Продолжение кода на следующей странице
void loop()
{
  // Получаем результат влажности в переменной
  int moisture= analogRead(A1);

  Serial.println(moisture);
}
```

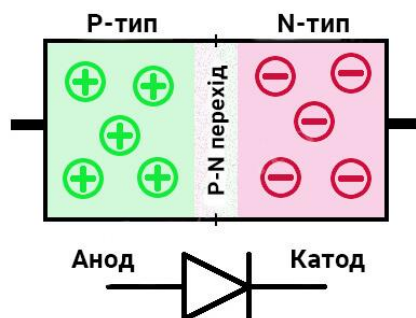
В общем, датчик освещенности выводит в бинарные данные, как «0» когда нет световых лучей или «1» когда есть световые лучи.

Также есть аналог датчика освещения – фоторезистор. По свойству он тоже такой, но относится не к датчикам, а к резисторам, таким как: потенциометр; резистор; тому подобное. Резисторы служат для изменения силы тока, а датчики служат для изменения напряжений. Вот так сказывается фоторезистор на схемах:



Также есть аналог фоторезистора и датчика освещения – фотодиод, относящийся к диодам. Диод – электронный компонент, пропускающий

электрический ток только в одну сторону – от анода до катода. Диод также называют выпрямителем, так как он превращает переменный ток в пульсирующий постоянный. Полупроводниковый диод состоит из пластинки полупроводникового материала (кремний или германий), одна сторона пластинки – с электропроводностью р-типа, то есть принимает электроны. Другая сторона отдает электроны, у нее проводимость n-типа. На наружные поверхности пластины нанесены контактные металлические слои, к которым припаяны проводные выводы диодных электродов.



Вот так обозначают фотодиод на схемах:



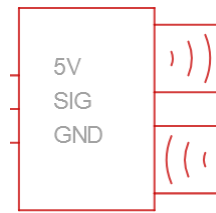
Также есть один из резисторов – гибкий сенсор, изменяющий напряжение при деформации этого резистора. Его схема:



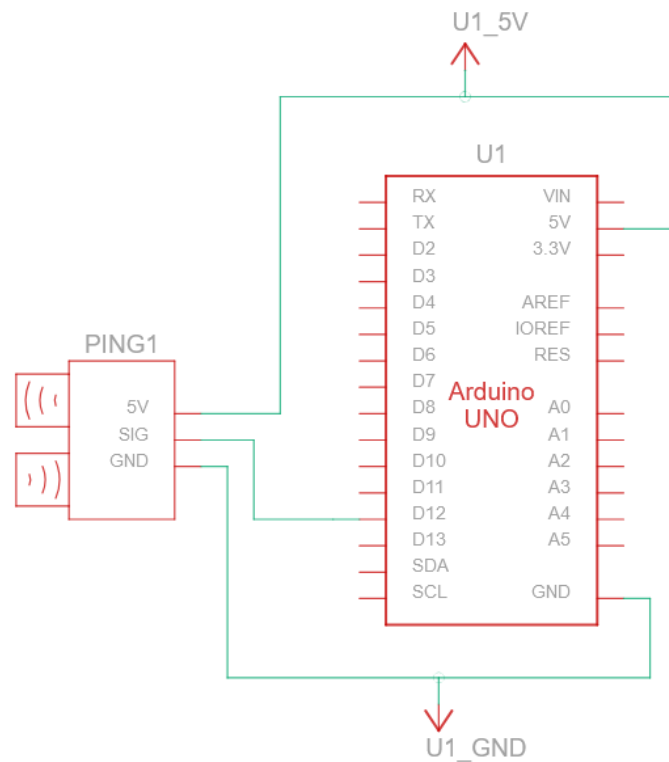
Также есть из резисторов – датчик склона, изменяющий напряжение при изменении склона датчика. Его схема:



Еще есть ультразвуковой датчик расстояния, относящийся к датчикам. Ультразвуковой датчик расстояния – датчик, использующий звуковые волны для определения расстояния до объекта. Он распознает объект от 200 и менее сантиметров. Его схема:



Теперь сделаем код для Arduino, который будет нам показывать расстояние какого-либо объекта от датчика. Вот схема:



Теперь сделаем код для этой схемы:

```
int sig = 0;

// Вот эту функцию нужно сделать
// triggerPin - пин для отправки импульса
// echoPin - пин для приема эха
long readUltrasonicDistance(int triggerPin, int echoPin)
{
    pinMode(triggerPin, OUTPUT);
    digitalWrite(triggerPin, LOW);
    delayMicroseconds(2);
    digitalWrite(triggerPin, HIGH);
    delayMicroseconds(10);
    digitalWrite(triggerPin, LOW);
    //продолжение кода на следующей странице
}
```

```

pinMode(echoPin, INPUT);

// Возвращает результат функции
return pulseIn(echoPin, HIGH);
}

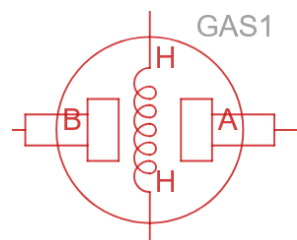
void setup()
{
  Serial.begin(9600);
}

void loop()
{
  // Перевод расстояния в сантиметрах
  sig= 0.01723 * readUltrasonicDistance(12,12);

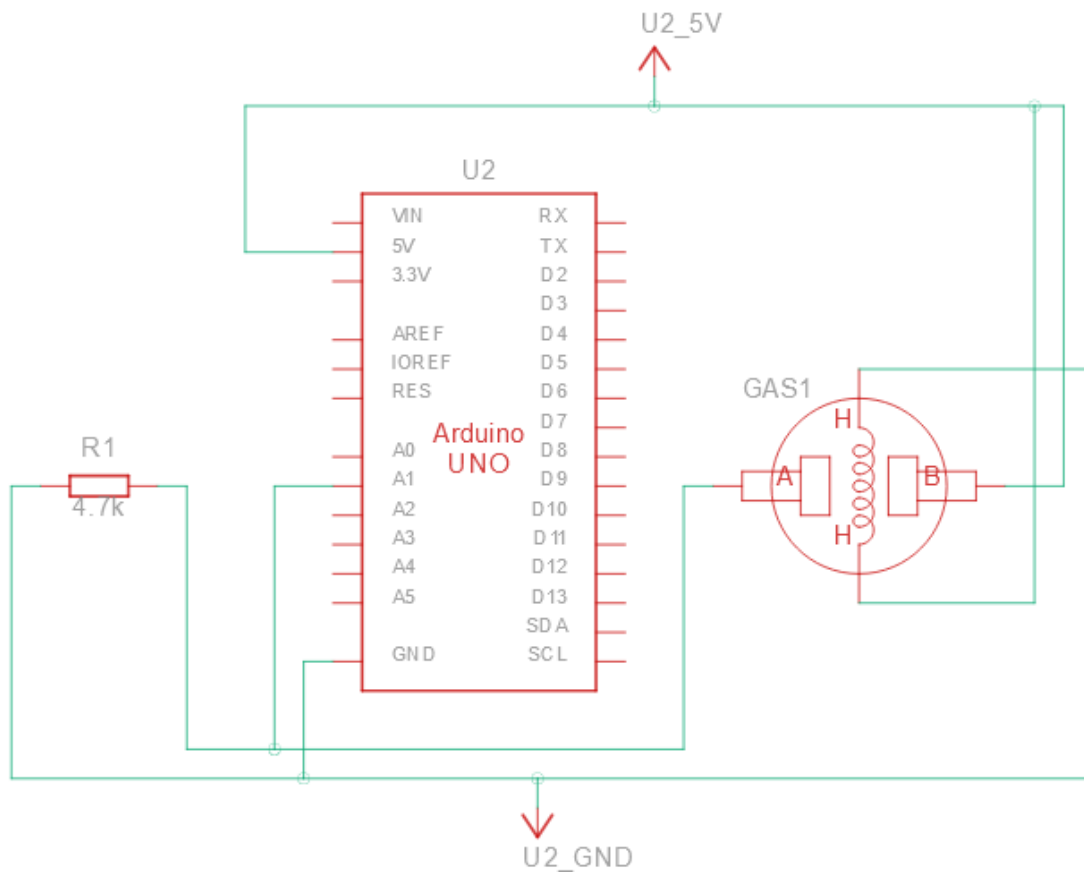
  // Показать результат в консоли
  Serial.println(sig);
}

```

Есть также датчик газа, который обнаруживает газ, такие как: LPG (сжиженные нефтяные газы); изобутан; метан; алкоголь; водород; дым; тому подобное. Вот как он выглядит на схемах:



Теперь сделаем код, в котором будут выводиться данные от датчика.
Схема:



Код:

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

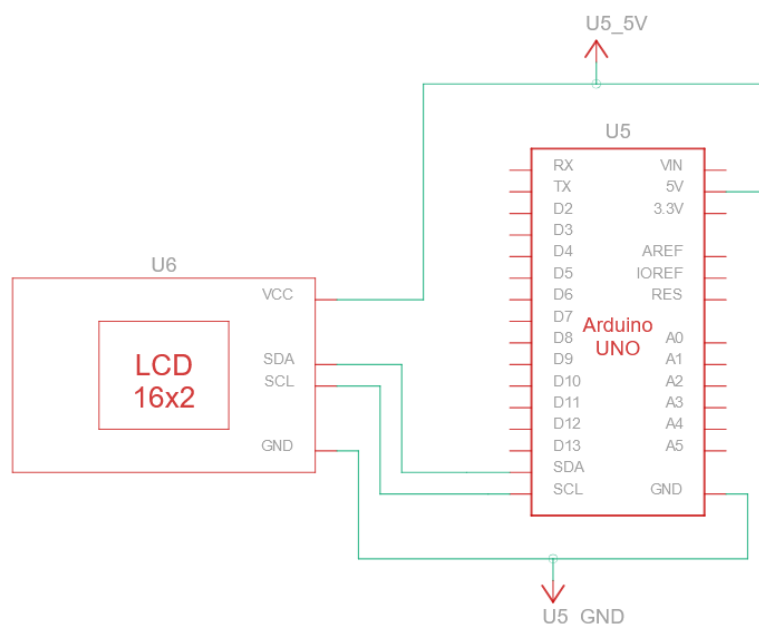
void loop()
{
  int moisture= analogRead(A1);
  Serial.println(moisture);
}
```

Урок 2.5. ИСПОЛЬЗОВАНИЕ ЖИДКО-КРИСТАЛЛИЧЕСКОГО ЭКРАНА.

Жидко-кристаллический экран – элементы радиоэлектроники, отображающие текст у себя на экране. Вот как сказывается этот элемент на схемах:



Теперь создадим код для этой схемы:



Если мы рассматривали элементы, не требующие импорта библиотеки, то в данном случае при использовании жидкокристаллического экрана следует импортировать библиотеку <Adafruit_LiquidCrystal.h>.

Вот код схемы:

```
#include <Adafruit_LiquidCrystal.h>

Adafruit_LiquidCrystal lcd_1(0);

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

void loop() //продолжение кода на следующей странице
```

```

{ // Продолжение кода на следующей странице
lcd_1.setCursor(0,0); // Установить координаты для
// показа текста
lcd_1.print("Hello World!"); // Показать текст

lcd_1.setCursor(0,1); // Установить координаты для
// показа значения переменной
lcd_1.print(seconds); // Показать значение переменной

lcd_1.setBacklight(1); // Включить подсветку экрана
delay(500);
  lcd_1.setBacklight(0); // Выключить подсветку экрана
  delay(500);
}

```

Урок 2.6. ИСПОЛЬЗОВАНИЕ 7-СЕГМЕНТНОГО ЭКРАНА.

7 сегментный экран – экран, состоящий из 7 сегментов на каждую цифру. Вот как он называется на схемах:



При сборке 7 сегментный экран также подсоединяются одинаково, как и жидкокристаллический экран. Это можно увидеть схему на стр. 60. Таким образом, можно тоже собирать радиотехнику с 7 сегментным экраном.

```

#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();
}

```

Обратите внимание, что на экране можно написать только следующие символы: «.»; «>»; "0"; "1"; "2"; "3"; "4"; "5"; "6"; "7"; "8"; "9"; «-»; "_".

