

C++

Для радіоелектроників

Вадим Савенко @ 2024

Зміст:

I РОЗДІЛ. ЗНАЙОМСТВО З C++ (стр. 3 – 44)

- 1.1. ЧОМУ ВАРТО ВИВЧАТИ C++ (стр. 3 – 7)
- 1.2. ЗМІННІ ТА ТИПИ ДАННИХ (стр. 7 – 17)
- 1.3. ОПЕРАТОРИ (стр. 17 – 24)
- 1.4. РОЗГАЛУДЖЕННЯ (стр. 24 – 31)
- 1.5. ЦИКЛИ (стр. 31 – 34)
- 1.6. ФУНКЦІЇ (стр. 34 – 43)
- 1.7. НАДАННЯ ЗМОГИ ВВЕДЕННЯ ДАНИХ КОРИСТУВАЧЕМ (стр. 43)
- 1.8. РАНДОМИ (стр. 44)

II РОЗДІЛ. ВИКОРИСТАННЯ ARDUINO З МОВОЮ C++ (стр. 45 – 61)

- 2.1. ЩО ТАКЕ ARDUINO (стр. 45)
- 2.2. ВИКОРИСТАННЯ РАДІОЕЛЕМЕНТІВ З ARDUINO (стр. 45 – 50)
- 2.3. ЩО ТРЕБА ЗНАТИ ПРИ НАПИСАННЯ КОДУ ДЛЯ ARDUINO (стр. 50 – 52)
- 2.4. ВИКОРИСТАННЯ ДАТЧИКІВ (стр. 52 – 59)
- 2.5. ВИКОРИСТАННЯ ЖИДКО-КРИСТАЛИЧНОГО ЕКРАНУ (стр. 60 – 61)
- 2.6. ВИКОРИСТАННЯ 7-СЕГМЕНТНОГО ЕКРАНУ (стр. 61)

І РОЗДІЛ. ЗНАЙОМСТВО З C++.

Урок 1.1. ЧОМУ ВАРТО ВИВЧАТИ C++.

Великий попит

До цього дня в різних галузях промисловості C++ є кращим вибором. Її широке використання в існуючих системах та адаптивність до нових викликів продовжують сприяти її популярності в різних галузях.

Переваги та Сила C++:

- **Висока Продуктивність:** C++ відомий своєю високою продуктивністю та ефективним використанням системних ресурсів.
- **Багата Стандартна Бібліотека:** він має велику стандартну бібліотеку, яка надає широкий спектр структур даних, алгоритмів і утиліт, що економить час і зусилля розробників.
- **Міцний Фундамент:** надає глибоке розуміння комп'ютерних наук, що робить його відмінною відправною точкою у світ програмування.
- **Навички:** багато концепцій, які ви вивчаєте в C++, застосовні до інших мов програмування, що полегшує вивчення додаткових мов у майбутньому.
- **Підтримка Спільноти:** він має велику та активну спільноту, що означає наявність великої кількості ресурсів, бібліотек і інструментів для розробників.

C++ - це мова програмування загального призначення, яка підтримує як **низькорівневу**, так і **високорівневу** парадигми програмування, що робить її дійсно адаптивним і потужним інструментом.

- **Низький рівень:** прямий контроль над апаратним забезпеченням, вимагає глибокого розуміння архітектури комп'ютера, **складніший, але точніший**.
- **Високий рівень:** більш зручне програмування, швидша розробка, легше обслуговувати і має більш **читабельний для людини код**.

Низький рівень	Високий рівень
<i>Переваги</i>	
повний контроль при приготуванні сендвіча з нуля	швидко та зручно, оскільки ви замовляєте сендвіч з оптимізованим приготуванням
<i>Недоліки</i>	
забирає багато часу і вимагає навичок та досвіду в приготуванні сендвічів	обмежена кастомізація, відсутність прямого контролю над процесом приготування сендвічів

Кожна програма на C++ **повинна мати** функцію `main()`. Синтаксис її виглядає наступним чином:

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

- `int main()`: початкова точка програми. Вона називається головною функцією, і саме з неї починається виконання програми.
- `{ }`: фігурні дужки визначають блок коду. Все, що знаходиться всередині цих дужок, належить до головної функції і є частиною логіки програми.
- `return 0;`: позначає кінець програми і вказує на те, що вона успішно виконалася. Значення 0 означає, що все пройшло добре. Якщо були проблеми, це значення може бути іншим у виведенні.

Зауважте! Оператор `return 0;` **не є обов'язковим** у кінці **головної функції**. Якщо його не вставити, компілятор автоматично додасть його.

Стандартні бібліотеки слугують як **сховища** заздалегідь написаного, багаторазового **коду**, що спрощує виконання типових завдань. Вони заощаджують розробникам **час і зусилля**, пропонуючи добре перевірені, стандартизовані інструменти для створення програмного забезпечення.

Уявіть, що ви використовуєте бібліотеку **як будівлю з готових великих блоків**. Це набагато швидше і зручніше, ніж створювати маленькі блоки з нуля і використовувати їх.

Чому варто використовувати стандартні бібліотеки та файли

Написання коду за допомогою бібліотеки схоже на написання книги за допомогою словника. Ми можемо легко замінити фразу лише одним словом без втрати основного сенсу. Наприклад:

Текст	Інший текст у якому було замінені слова на фрази
<i>У біологічній спільноті, де організми взаємодіють з фізичним оточенням, все повинно бути в такому стані, в якому різні елементи рівні або у правильних пропорціях.</i>	<i>У <u>екосистемі</u> все повинно бути в <u>рівновазі</u></i>

Директиви препроцесора

Щоб додати зовнішні файли до вашої програми, вам потрібно використовувати директиви препроцесора. Це команди, які керують **препроцесором, інструментом, який перетворює код перед компіляцією**. Синтаксис більшості директив препроцесора такий:

```
#директива параметр
```

- **#** - символ, який вказує на те, що це директива препроцесування.
- **директива**: конкретна директива препроцесування;
- **параметр**: пов'язане значення або аргумент для цієї директиви.

Команда, яка додає зовнішні файли до вашої програми, називається `#include`.

```
#include <ім'я>
```

Зауважте! Стандартні файли приєднуються за допомогою кутових дужок `< >`, але ви також можете створювати **власні файли** і приєднувати їх до вашого проекту аналогічно, використовуючи подвійні лапки `" "`.

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

Як працює `#include`

Подивіться на код нижче і спробуйте запустити його.

Перед нами два файли: **main.cpp**; **header.h**.

Код **main.cpp**:

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

Код **header.h**:

```
}
```

Ви отримаєте *помилку* про відсутність `}`. Це зроблено навмисно, щоб показати, як працює `#include`. Ми можемо створити окремий файл, що

містить лише символ `}` і **включити** його у файл *main.cpp* за допомогою директиви `#include`.

Традиційною першою програмою на будь-якій мові програмування часто є проста програма, яка виводить повідомлення на консоль.

Введення та Виведення

Найпоширеніший спосіб виведення інформації у консоль - це використання бібліотеки `iostream`. Вона означає **вхідний(input)/вихідний(output)** потік.

Як ми вже знаємо, щоб додати бібліотеку до нашої програми, ми повинні використати `#include`.

```
#include <iostream>
```

Виведення символів

Потік `cout` є частиною бібліотеки `iostream` і використовується для **стандартного виведення**.

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

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

- `std::` означає, що ідентифікатор є частиною стандартної бібліотеки, яка організована в просторі імен **std**.
- `cout`: відповідає за **виведення символів** і використовується для відправлення виводу.
- `<<`: використовується для вставки даних у стандартний потік виводу.

Коментарі

Коментарі використовуються для документації та не впливають на виконання програми. Існує два типи:

- **Однорядкові коментарі**: Починаються з `//` та простягаються до кінця рядка.
- **Багаторядкові коментарі**: Заключенні між `/*` і `*/`.

Завдання. Створіть програму, яка виводить ваше повідомлення.

Для розв'язку цього завдання ми повинні використати бібліотеку `iostream` та використати `main()`. Також треба використати команду для відображення інформації:

```
std::cout << "Повідомлення";
```

Тоді ми отримуємо код:

```
#include <iostream>

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

Та при виконання дій коду ми отримуємо результат у консолі:

```
Повідомлення
```

Урок 1.2. ЗМІННІ ТА ТИПИ ДАНИХ.

У сфері мов програмування, змінні відіграють важливу роль у **зберіганні та маніпулюванні** даними. Процес створення змінної включає два необхідних кроки: оголошення та ініціалізацію.

- **Оголошення:** процес **введення змінної** компілятору. Він передбачає вказівку типу даних змінної і, за бажанням, надання їй імені.
- **Ініціалізація:** процес **присвоєння значення** оголошеній змінній. Це критично важливо, щоб уникнути потрапляння непередбачуваних або сміттєвих значень до змінної.

```
#include <iostream>

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

Примітка! Ви можете вказати початкове значення для змінної під час оголошення.

Також при оголошенні декількох змінних, **які містять дані одного типу**, зручно вказувати тип даних лише один раз. Наприклад:

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

Конвенції іменування

Вибір **значущих та описових** імен для змінних є важливим для написання зрозумілого та легкого для підтримки коду.

```
int n; // Погане іменування
```

АБО

```
int kilkistStudentiv; // Гарна назва змінної
```

Назви змінних у C++ можуть складатися з літер і цифр, але не можуть починатися з цифр, також вони не можуть містити пробіли чи арифметичні оператори. Також **уникайте використання** назв, які збігаються з **зарезервованими ключовими словами**. Це може призвести до компіляції *помилки*.

Так йменувати змінні не можна як ось у цьому коді:

```
int if = 25;           // Використовує зарезервоване слово  
int email@number = 25; // Містить спеціальний символ  
int my age = 37;       // Містить пропуск  
int 121A = 121;        // Починається з цифри
```

При оголошенні змінної потрібно вказати, який тип даних ми будемо в ній зберігати. Для зручності роботи з пам'яттю існують типи даних для кожної ситуації.

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

Числові

Ці типи необхідні для зберігання числових значень та маніпулювання числовими даними. Ми можемо розділити їх на дві групи:

- Типи даних для цілих чисел (наприклад, 12);
- Типи даних для чисел з плаваючою комою (наприклад, 0.12).

Булеві

Тип даних `bool` представляє два логічних значення: **нуль** інтерпретується як *false* та **один** інтерпретується як *true*.

Світло увімкнено = *true* = 1 Світло вимкнено = *false* = 0

Char

Тип даних `char` використовується для зберігання окремих символів, які можуть включати літери, цифри, знаки пунктуації та спеціальні символи.

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

Цілі числа

Тип даних `int` може зберігати значення в діапазоні від 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;
}
```

Це відбувається тому, що коли ви оголошуєте змінну типу `int`, вона виділяє рівно 4 байти пам'яті вашого комп'ютера. А числа вище 2147483647 (або нижче -2147483648) не поміщаються у ці 4 байти. На щастя, інші доступні типи даних можуть виділити вам більше (або менше) місця для ваших потреб. Ось таблиця:

Тип даних	Діапазон	Розмір
short	-32,768 - 32,767	2 байти
int	-2,147,483,648 до 2,147,483,647	4 байти
long	-9,223,372,036,854,775,808 до 9,223,372,036,854,775,807	8 байт

Отже, ви можете використовувати `long` для зберігання великих чисел (наприклад, населення світу). Ви також можете використовувати `short`, якщо ви впевнені, що ваше число **не вийде за межі діапазону** від -32,768 до 32,767 (наприклад, для зберігання віку користувача). Використання `short` займе менше місця.

```
#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;
}
```

Зауважте. Будьте обережні з типом даних, який ви обираєте. Якщо діапазон типу *перевищено*, компілятор C++ **не скаже** вам про це, і ви отримаєте неочікуване значення **не знаючи, що щось не так**.

Числа з плаваючою комою

Наведені вище типи даних призначені для зберігання цілих чисел. Якщо ми спробуємо присвоїти 1.6 одному з них, ось що ми отримаємо:

```
#include <iostream>

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

Тип `int` ігнорує десяткову частину числа. Така ж історія з `short` або `long`. Для зберігання чисел з плаваючою комою (десяткових) слід використовувати тип даних `float` або `double`.

Тип Даних	Точність	Розмір
<code>float</code>	7 десятичних знаків	4 байти
<code>double</code>	15 десятичних знаків	8 байтів

Ось приклад використання `double` для зберігання `1.6`.

```
#include <iostream>

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

Зауважте. Оскільки тип `float` має точність лише 7 цифр, число `123.456789` вже виходить за межі його діапазону. Це може призвести до неточних результатів (як показано у прикладі нижче). Тому краще використовувати `double` за замовчуванням, якщо ви не впевнені, що точність `float` достатня.

```
#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, що ви можете використовувати `float` або `double` для зберігання цілих чисел, оскільки це десяткові числа з десятичною частиною, що дорівнює 0. Однак, як правило, якщо змінна зберігає значення, які можуть бути лише цілими числами (наприклад, населення або вподобання), слід використовувати `short/int/long`.

```
#include <iostream>

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

Ви також можете зберігати **текст** у змінних, окрім символів за допомогою char. Для цього потрібно

- Підключити файл string
- Використовувати розширення області видимості std
- Оголосити змінну типу string

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

int main()
{
    //declaring string variable
    std::string myStringVariable = "codefinity";

    //displaying our string variable
    std::cout << myStringVariable << std::endl;
}
```

Рядкові змінні також можуть містити числа (у вигляді тексту):

Якщо ви спробуєте додати дві змінні рядка, ви отримаєте **конкатенацію** (це працює без пробілів):

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

int main()
{
    std::string var1 = "Hello "; //space is also a symbol
    std::string var2 = "World";

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

Те саме станеться з числами – вони не будуть додані алгебраїчно.

Масив - це **набір елементів одного типу**. Щоб створити масив, слід виконати такі дії:

- Визначте **тип даних** для елементів, які ви збираєтеся зберігати в масиві;
- Присвоїти масиву **ім'я**;
- Вказати кількість елементів у масиві, помістивши цю кількість у **квадратні дужки** після його імені. Наприклад:

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

Компілятор видасть помилку, якщо розмір не вказано у статичних масивах.

Для ініціалізації масиву потрібно **вказати всі його елементи у фігурних дужках**:

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

Щоб отримати потрібний нам елемент з масиву, ми можемо посилатися на нього за допомогою **індексів**. Кожен елемент масиву має свій індекс, так само як кожен будинок у вашому місті має свою адресу.

Зауважте. Індекс починається з індексу 0, а не з 1.

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

Довжина масиву, наведеного вище, дорівнює 6. Якщо ми створимо масив довжиною 5 з цими числами, то він видасть помилку.

```
#include <iostream>

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

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

Припустимо, що в масиві є **більше** елементів, ніж ви вказали при оголошенні. У цьому випадку виникне **помилка** компіляції**, оскільки

компілятор виділяє фіксовану кількість пам'яті при оголошенні масиву. Це все одно, що намагатися налити більше води у вже повну склянку.

Якщо в масиві **менше** елементів, ніж ви вказали при оголошенні, то всі неініціалізовані елементи **будуть дорівнювати нулю** або матимуть **сміттєві значення** (непередбачувані або довільні дані).

```
#include <iostream>

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

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

Ви можете уявити масив як книгу, в якій кожна сторінка (елемент) пронумерована (індекс). Дані в масиві можна змінювати, для цього потрібно звернутися до елемента за індексом *i*, наприклад, задати йому нове значення:

```
#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;
}
```

Масив може бути елементом іншого масиву, наприклад, оголосимо масив, елементами якого будуть інші масиви. Щоб оголосити **багатовимірний масив**, вам знадобиться ще одна пара квадратних дужок:

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

- Перша пара дужок - це основний масив;
- Друга пара дужок говорить про те, що елементами основного масиву будуть малі масиви.

```
#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;
}
```

Ми створили масив під назвою myArray, який містить чотири елементи, кожен з яких є масивом з трьома елементами.

Розмір змінної - це обсяг пам'яті, зарезервований компілятором. Компілятор резервує певну кількість байт з пам'яті вашого комп'ютера, виходячи з типу даних, який ви використовуєте. Ви можете скористатися функцією `sizeof()`, щоб дізнатися розмір змінної або типу даних у байтах. Наприклад:

```
#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++ дозволяє вибрати тип з точним розміром біт, наприклад, `int8_t`, `uint8_t`, `int16_t`, `uint16_t` тощо. Щоб використовувати ці типи даних, вам потрібно включити заголовний файл `<cstdint>`.

```
#include <cstdint>
```

Крім того, ми можемо змусити компілятор визначити тип змінної самостійно за допомогою ключового слова `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++ також дозволяє вам перейменувати існуючі типи даних під себе. Для цього використовується `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;
}
```

Під час компіляції рядок `typedef` повідомляє компілятору, що `MY_NEW_TYPE` - це просто тип `double`.

Урок 1.3. ОПЕРАТОРИ.

Оператор **присвоєння** (=): в програмуванні використовується для присвоєння значення змінній. **Синтаксис** виглядає наступним чином:

```
#include<iostream>

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

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

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

З типом даних `string` це працює точно так само:

```
#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;
}
```

Оператори **рівність** (==) і **нерівність** (!=) використовуються для чисельного порівняння 2 змінних:

```
#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;}
}
```

Чому 1 і 0? Це альтернативний підхід до використання булевого типу даних. Коли вираз `var == 9` є `true`, він представляється як 1, і це означає, що `var` дійсно дорівнює числу 9. І навпаки, коли вираз `var == -9` має значення `false`, він представляється як 0, що означає, що `var` не дорівнює числу -9.

Оператор **нерівність** (`!=`) діє з точністю до навпаки:

```
#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;
}
```

Ці п'ять математичних операторів (+, -, *, / і %) служать для виконання різних математичних операцій:

```
#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;
    // Продовження коду у наступній сторінці
}
```

```

//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 залишку від ділення (**modulo**)(%) обчислює і повертає залишок від стандартної операції ділення.

Оператор ділення (/) повертає лише цілу частину результату, відкидаючи залишок. Наприклад, при діленні 10 на 3 результатом буде 3, а не 3.333... Щоб отримати потрібний результат ділення з десятковими знаками (наприклад, $10 / 3 = 3.333$), необхідно, щоб хоча б один з операндів мав тип даних `double` або `float`.

```

#include <iostream>

int main()
{
    // one of the variable 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;
}

```

Оператор **сума** є єдиним математичним оператором, який можна застосувати до рядка (він називається **конкатенація**):

```
#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;
}
```

Існують також **оператори порівняння** (>, <, <=, >=). Вони використовуються, коли вам потрібно чисельно **порівняти** значення з певним діапазоном:

```
#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 одночасного обчислення декількох умов ви повинні використовувати логічні оператори **НЕ** (!), **ТА** (&&) та **АБО** (||). Давайте проілюструємо використання логічних операторів у реальних сценаріях:

- Я візьму гаманець на прогулянку, якщо на моєму шляху є банк **ТА** магазин;

- Я не візьму гаманець на прогулянку, якщо по дорозі буде **НЕ** банк;
- Я візьму гаманець на прогулянку, якщо по дорозі є банк **АБО** магазин.

```
#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;

    //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 << "There is a bank or a wallet = " << wallet2 << std::endl;
}
```

Складений оператор присвоювання - це комбінований оператор, який поєднує в собі оператор присвоювання та арифметичний (або побітний) оператор. Результат такої операції присвоюється **лівому операнду** (тому, що знаходиться **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;
}
```

Використання іншого складеного оператора присвоєння:

```
#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;

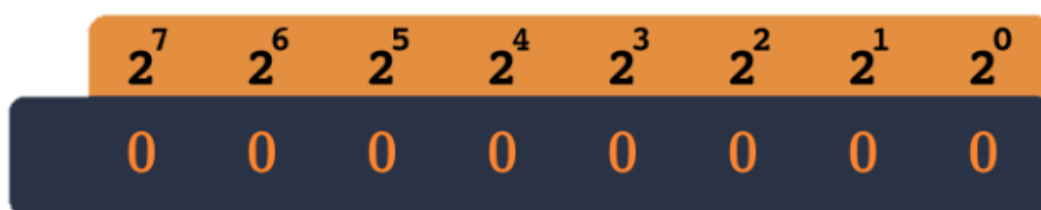
    value *= 4; // value = value * 4
    std::cout << "value = 16 * 4 = " << value << std::endl;

    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;
}
```

Що таке байт

Це **байт**, фундаментальна одиниця зберігання цифрової інформації. Байт складається з **8 бітів**.



В основі зберігання цифрової інформації лежить байт, що складається з 8 бітів. Кожен біт представляє собою цифру, а числа зазвичай виражаються у двійковій системі числення. Двійкове представлення передбачає вираження **числа як суми степенів 2**.

Приклади двійкового представлення

$$1010 = 10$$

$$111111 = 255$$

$$101010 = 42$$

Розглянемо число 4, яке у двійковій системі числення має вигляд *00000100*. Кожен біт у цьому байті відповідає степеню 2:

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

Аналогічно, число 5 представляється як *00000101*:

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

Bitwise shift

Порозрядний зсув

- У цьому контексті **один зсув вліво**: значення **множить значення на 2**.
- І навпаки, коли ми виконуємо **один бітовий зсув вправо**: значення **ділиться на 2**.

```
#include <iostream>

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

Вважайте **операцію зсуву праворуч та ліворуч** >> АБО << на *n* позицій як ділення або ж множення на **2ⁿ**.

У випадку $10 \gg 3$ це фактично ділення 10 на 2^3 (що дорівнює 8), а результат дорівнює 1.

Математичні оператори, необхідні для наступної задачі

Оператор	Опис
-	Віднімає одне значення від іншого
*	Множить два значення
/	Ділить одне значення на інше
+	Додає одне значення до іншого
%	Показує залишок при діленні одного значення на інше
**	Возвести до степеня
//	Ділення націло

Урок 1.4. РОЗГАЛУДЖЕННЯ.

Конструкція `if...else` у програмуванні дозволяє вашій програмі обирати різні шляхи та керувати різними потенційними результатами.

Вона складається з двох основних компонентів: умови і відповідних дій або наслідків на основі цієї умови.

Ось ілюстрація:

```
#include<iostream>

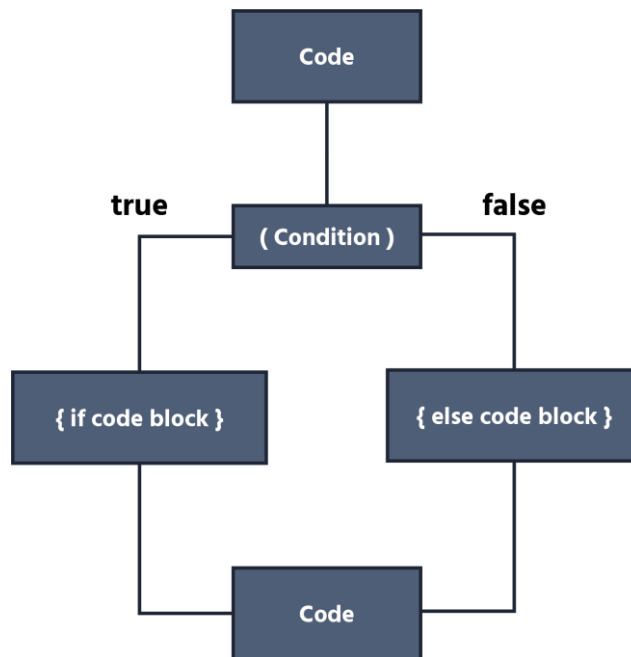
int main()
{
    int var = 13;

    /* 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;
    }
}
```

Існує також обробка **протилежного** випадку за допомогою `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;
    }
}
```

Усередині `if...else` можуть бути інші `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;
        }
    }
}
```

Ви також можете використовувати конструкцію `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 **термінальний оператор** пропонує лаконічну альтернативу оператору `if...else`, з помітною відмінністю. Він складається з трьох ключових елементів:

1. Булевого виразу;
2. Інструкції для випадку істина;
3. Інструкції для випадку false.

```
(булевий вираз)? інструкція_для_істинного_випадку : інструкція_для_хибного_випадку
```

Також можна робити таким чином для отримання результату у консолі:

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

int main()
{ //продовження коду у наступній сторінці
```

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

Такий оператор зручно використовувати, наприклад, при порівнянні двох чисел:

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

У цьому випадку результат потрібної операції було присвоєно змінній `result`.

Коли порівняння повертає **істинний** результат, значення `var1` буде збережено у змінній `result`.

if true then

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



І навпаки, якщо результат порівняння **false**, змінній `result` буде присвоєно значення змінної `var2`.

if false then

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



Зауважте. Дотримуйтесь сумісності типів даних!

Як би це виглядало з використанням `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;
}
```

Конструкція **switch-case** дозволяє порівнювати результат виразу з набором наперед визначених значень. Структура switch-case:

```
switch (someExpression)
{
    case someCheck1:
        // якщо цей випадок дозволений
        зробити_щось1;
        break;

    case someCheck2:
        // якщо цей випадок дозволений
        зробити_щось2;
        break;

    за замовчуванням:
        //якщо жоден з випадків не підходить
        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 - це оператор, який означає вихід із блоку коду;
- default - це необов'язкова, але корисна частина. Ця частина буде виконана, якщо жоден із варіантів не підійде.

У нашому випадку ми перевіряємо змінну, якщо вона дорівнює 5, тоді буде відображено відповідний текст, і, використовуючи оператор break, потік програми вийде з усієї конструкції switch-case, і не буде обробки інших випадків.

- break - оператор означає вихід з блоку коду.
- default - необов'язкова, але корисна частина. Ця частина буде виконана, якщо жоден з випадків не підходить.

У нашому випадку ми перевіряємо змінну, якщо вона дорівнює 5, то виводиться відповідний текст і за допомогою оператора break потік програми покине всю конструкцію switch-case, і не буде ніякої обробки інших випадків.

```
#include <iostream>

int main()
{
    int variable = 5;

    switch (variable) //продовження коду у наступній сторінці
```

```

{
    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;"
}
}

```

Але оператор "перемикання" має одне застереження. Ми навмисно вилучаємо оператор break:

Ми використовували if...else, switch-case для порівняння наших змінних з іншими значеннями. Але що, якщо нам потрібно зробити щось сто разів? Тисячу разів? Мільйон разів?

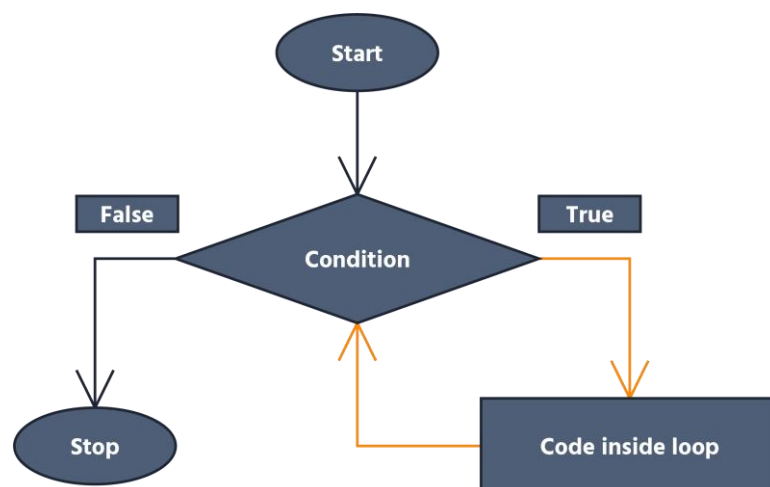
Саме для таких випадків і призначені **цикли**! Вони дозволяють зациклити вашу програму за певних умов. Структура циклу while:

Урок 1.5. ЦИКЛИ.

```

while (someExpression == true)
{
    // якщо 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;
}

```

У цьому випадку ми підраховували ($x+=1$) **992** разів. Цикл виконувався до тих пір, поки $x+y$ не стало рівним результату (1000).

Як тільки вираз $x + y$ став рівним результату, цикл завершився, і ми отримали корінь рівняння (x).

Зауважте. Якщо умова не виконується, цикл може не розпочатися.

Важливо переконатися, що цикл має умову виходу, тобто що цикл не буде нескінченним. Приклад нескінченного циклу:

```

#include <iostream>

int main()
{
    bool condition = true;

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

```


На відміну від циклу `while`, який може ніколи не виконатися, цикл **do...while** гарантовано виконається **принаймні один раз**. Структура циклу `do...while`:

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

Зауважте. Рядок, що містить частину `while`, закінчується крапкою з комою (;).

Цикл **for** складніший за інші цикли і складається з трьох частин. Структура циклу `for`:

```
for (лічильник; умова виходу; вираз циклу)
{
    // блок інструкції
    зробити_щось;
}
```

- Лічильник;
- Умова виходу;
- Вираз циклу.

```
#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.

Тепер питання, Скільки ітерацій зробить цей цикл?:

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

А) 5;

Б) 6;

В) 1;

Г) 0;

Урок 1.6. ФУНКЦІЇ.

Функції - це невеликі підпрограми, які можна викликати за потреби.

Кожна функція має ім'я, за яким її можна викликати.

```
int main() // "main" - це ім'я функції
{
    return 0
}
```

Зауважте. Ім'я main вже зарезервовано мовою C++. Тому при оголошенні функції з таким іменем компілятор видасть помилку.

Щоб створити функцію, потрібно:

- визначити **тип** даних, які вона буде повертати;
- присвоїти їй **ім'я**;
- надати блок інструкцій (**тіло**) у фігурних дужках { . . . } для визначення її функціональності.

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

Наприклад, створимо функцію, яка виводить текст "c<>definity":

```
std::string nameOfCourses() // тип та ім'я функції
{ // початок тіла

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

} // кінець тіла
```

```
#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;
}
```

Створімо функцію, яка спрощує процес перетворення температур з Фаренгейта на Цельсія. Це практичне застосування у повсякденному житті.

```
#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;
}
```

Примітка! Аргумент функції представлений змінною `degree`, яка містить дані, з якими функція працює. У цьому контексті вона відноситься до температур у градусах Фаренгейта, які потрібно перетворити у градуси Цельсія. Пізніше ми детальніше розглянемо **аргументи функцій**.

Компілятор обробляє наш програмний код послідовно, подібно до того, як людина читає книгу, і якщо він зустрічає невідомі імена змінних чи функцій, він виведе помилку.

Для прикладу, спробуємо викликати функцію до її визначення.

Цей приклад викликає помилку. Це зроблено навмисно.

```
#include <iostream>

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

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

У таких ситуаціях важливо використовувати прототипи функцій.

Мета прототипу полягає в тому, щоб заздалегідь повідомити компілятор про нашу функцію. Створення прототипу подібне до стандартного оголошення функції, але з тонкою різницею:

- вказати **тип** майбутньої функції;
- дати їй **ім'я**;
- **аргументи** (за потреби);
- поставити знак кінця виразу.

```
int FahrenheitToCelsius(int degree); //прототип нашої функції
```

Додамо прототип функції до нашого прикладу, який викликав помилку:

```
int FahrenheitToCelsius(int degree); //прототип нашої функції
```

Примітка! Прототипування є корисним, коли ви працюєте з великою кількістю функціоналу. Щоб уникнути "сміття" в основному файлі, прототипи та визначення функцій переносяться до сторонніх файлів і підключаються до основного файлу з допомогою директиви `#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;
}

```

Примітка! Прототипування є корисним, коли ви працюєте з великою кількістю функцій. Щоб уникнути "сміття" у головному файлі, прототипи і визначення функцій перенесено до сторонніх файлів та включені в основний файл за допомогою директиви `#include`.

При створенні функції завжди вказується тип даних, які вона повертає.

У прикладі функції `main` ми бачимо, що вона має цілочисельний тип даних, а це означає, що після завершення роботи вона поверне ціле значення, в нашому випадку число 0.

```

int main()
{
    return 0;
}

```

Зауважте. Оскільки функція `main` є зарезервованою у C++, вона завжди повертатиме ціле число.

Але наші функції можуть повертати будь-яке значення:

```

double notMainFunc()
{
    return 3.14;
}

```

Щоб викликати функцію, потрібно написати її ім'я в дужках:

```
notMainFunc();
```

```
#include <iostream>

double notMainFunc()
{
    return 3.14;
}

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

Ми створили функцію, яка **повертає** значення 3.14 як `double` **тип даних**, і ми викликали цю функцію, щоб показати її вивід на екрані.

Функції також можуть бути `string` типу:

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

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

`typedef` також можна застосовувати:

```
#include <iostream>

typedef int MY_NEW_TYPE;

MY_NEW_TYPE TYPEfunc() //MY_NEW_TYPE function
{
    return 777; //продовження коду у наступній сторінці
}
```

```

}

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

```

Якщо ви не можете точно вказати тип повернення, оператор `auto` змусить компілятор зробити це 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` завершує виконання функції і повертає значення **визначеного типу**.

```

int func() //int - наперед визначений
{
    змінна int = 10;
    повернути змінну; //змінна = 10
}

```

Якщо тип вказано неправильно, функція буде поводитись непередбачувано:

```

#include <iostream>

unsigned short func()
{
    return -10; //продовження коду у наступній сторінці
}

```

```

}

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

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

```

Тобто, перед створенням функції **необхідно вказати тип даних, які вона повертає**. Також в C++ існують спеціальні функції **void**. Функції з таким типом даних можуть не повертати нічого або повернути "нічого".

```

#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();
}

```


Зауважте. Зазвичай функції типу void просто виводять статичний текст або працюють з вказівниками

```
#include <iostream>

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

    return a;
    return b;
}

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

Функція з параметром (**аргументом**) - це функція, яка має працювати з об'єктом "зовні".

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

Примітка. Аргумент функції - це локальна змінна, яка створюється та існує лише у поточній функції.

Наприклад, давайте створимо функцію, яка ділить будь-яке ціле число на 2:

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

Щоб використати таку функцію, потрібно викликати її і передати їй аргумент:

```
func(10);
```

Наведемо приклад:

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

У функцію можна передавати декілька аргументів:

Наприклад, створимо функцію, яка ділить будь-яке ціле число на 2:

`func(5, 7)` - виклик функції і передача їй аргументів. Аргументи передаються через кому (,).

Також можна передавати масиви:

Щоб використати таку функцію, ви повинні викликати її і передати в неї аргумент:

```
func(10);
```

Ось приклад:

```
#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);
}
```

Ви можете передавати декілька аргументів у функцію:

```
#include <iostream>

int func(int a, int b)
{
    return a + b; //the function to sum arguments
} //продовження коду у наступній сторінці
```

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

`func(5, 7)` – виклик функції та передача аргументів до неї. Аргументи передаються через коми (,). Ви також можете передавати масиви:

```
#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. НАДАННЯ ЗМОГИ ВВЕДЕННЯ ДАНИХ КОРИСТУВАЧЕМ.

Якщо ми знаємо про `std::cout` для відображення тексту, але ще можна питати користувача, щоб він написав дані для змінної за допомогою команди `std::cin`. Такий код виглядає таким чином:

```
#include <iostream>

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

Урок 1.8. РАНДОМИ.

Для знаходження випадкового числа, який видасть комп'ютер, треба зробити рандомність чисел у коді. Такий код виглядає таким чином:

```
#include <iostream>

int main()
{
    int n = /*Від якого числа буде рандом*/;
    int m = /*До якого числа буде рандом*/;

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

    /*Показати рандомне число у консолі*/
    std::cout << random << std::endl;
}
```

II РОЗДІЛ. ВИКОРИСТАННЯ ARDUINO З МОВОЮ C++.

Урок 2.1. ЩО TAKE ARDUINO.

Arduino (Ардуіно) — апаратна обчислювальна платформа для аматорського конструювання, основними компонентами якої є плата мікроконтролера з елементами вводу/виводу та середовище розробки Processing/Wiring на мові програмування, що є спрощеною підмножиною C/C++. Arduino може використовуватися як для створення автономних інтерактивних об'єктів, так і підключатися до програмного забезпечення, яке виконується на комп'ютері (наприклад: Processing, Adobe Flash, Max/MSP, Pure Data, SuperCollider). Інформація про плату (малюнок друкованої плати, специфікації елементів, програмне забезпечення) знаходяться у відкритому доступі і можуть бути використані тими, хто воліє створювати плати власноруч.

Назва Arduino походить від бару в Івреа, Італія, де зустрічалися деякі із засновників проекту. Бар був названий на честь Ардуїн I, який був маркграфом Маршу Івреї та королем Італії з 1002 по 1014 роки.

Урок 2.2. ВИКОРИСТАННЯ РАДІОЕЛЕМЕНТІВ З ARDUINO.

Для вивчення використання радіоелементів з Arduino, треба спочатку дізнатись про використання радіоелементів без Arduino.

Радіoeлектроніка — галузь науки і техніки, яка охоплює теорію, методи створення та використання пристроїв для передавання, приймання та перетворення інформації за допомогою електромагнітної енергії.

Термін «радіoeлектроніка» з'явився в 50-х роках 20 століття і є до певної міри умовним. Радіoeлектроніка охоплює радіотехніку і електроніку, в тому числі напівпровідникову електроніку, мікроелектроніку, квантову електроніку, ІЧ техніку, хемотроніку, оптоелектроніку, акустoeлектроніку, кріoeлектроніку та інші радіoeлектроніка тісно пов'язана, з одного боку, з радіофізикою, фізикою твердого тіла, оптикою і механікою, а з іншого — з електротехнікою, автоматикою, телемеханікою і обчислювальною технікою.

Методи та засоби радіoeлектроніки знаходять широке застосування у радіозв'язку, космічній техніці, системах дистанційного керування, радіонавігації, автоматичній, обчислювальній техніці, радіолокації,

військовій техніці та спеціальній техніці, в побутовій техніці тощо. Продукцією радіоелектроніки є радіоелектронна апаратура.

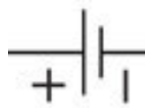
Сфера використання радіоелектроніки безперервно розширюється, проникаючи в економіку, промислове виробництво, сільське господарство, медицину, транспорт та в інші області людської діяльності.

Для подання сигналу у електроніці потрібно джерело живлення. Загалом джерелом живлення може бути: батарея; акумулятор; вилка з проводом; тощо. Кожне живлення має заряди, такі як «Анод» та «Катод». Катод - електрод деякого приладу, приєднаний до від'ємного полюса джерела струму, а анод - електрод, від якого в газі чи електроліті напрямлений струм (рух позитивних зарядів).

У літературі зустрічаються різні позначення знаку катода — «-» або «+», що визначається, зокрема, особливостями описуваних процесів. У електрохімії прийнято вважати, що «-» катод — електрод, на якому відбувається процес відновлення, а «+» анод — той, де проходить процес окислення. Під час роботи електролізера (наприклад, під час рафінування міді) зовнішнє джерело струму забезпечує на одному з електродів надлишок електронів (від'ємний заряд), тут відбувається відновлення металу, це катод. На іншому електроді забезпечується нестача електронів і окислення металу, це анод. Разом з тим, під час роботи гальванічного елемента (наприклад, мідно-цинкового), надлишок електронів (і від'ємний заряд) на одному з електродів забезпечується не зовнішнім джерелом струму, а власне реакцією окислення металу (розчинення цинку), тобто в гальванічному елементі від'ємним, якщо дотримуватися наведеного визначення, буде анод. Електрони, проходячи через зовнішнє коло, витрачаються на проходження реакції відновлення (міді), тобто катодом буде додатний електрод. Так, на наведеній ілюстрації зображено позначений знаком «+» катод гальванічного елемента, на якому відбувається відновлення міді. Відповідно до такого тлумачення, для акумулятора знак анода і катода змінюється залежно від напрямку протікання струму.

В електротехніці за напрямком струму прийнято вважати напрямком руху додатних зарядів, тому у вакуумних і напівпровідникових приладах і електролізних комірках струм тече від додатного анода до від'ємного катода, а електрони, відповідно, навпаки, від катода до анода.








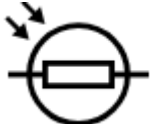
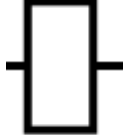


Загалом про створенні радіотехніки, завжди робиться схеми приладу. Ось так позначають джерело живлення:



У електроніці елементи поділяються для введення інформації (Такі як; резистори; діоди; датчики; тощо) та для виведення інформації (Такі як: світлодіоди; бузер; двигун; тощо). Світлодіод – елемент, який працює як лампочка. Бузер – елемент, який робить звуковий шум.

Ось таблиця, як позначаються елементи:

Елемент	Схематичне зображення
Джерело живлення	
Світлодіод	
Бузер	
Динамік	
Кнопка	
Постійний резистор	або
Змінний резистор (Потенціометр)	
Резистор із зазначенням потужності	
Діод	
Стабілітрон	

Конденсатор	
Полярный конденсатор	
Перемикач	
Двигун	
Індуктор	
Вольтметр	
Амперметр	
Фоторезистор	
Реле	
NPN-транзистор	
PNP-транзистор	

Тактову кнопку (іноді називають "пуш-батон") - це перемикач, який при натисканні замикає або розмикає електричне коло. Вона має два стани: замкнутий (під час натискання) і розімкнутий (коли не натиснута). Використовується для запуску або зупинки різних електронних пристроїв або для виконання певних команд.

Резистор - це компонент, що обмежує струм в електричному колі. Він має певний опір, який вимірюється в омах (Ω). Резистори використовуються для контролю струму, розподілу напруги і захисту інших компонентів від надмірного струму.

Потенціометр - це змінний резистор, опір якого можна регулювати вручну. Потенціометри часто використовуються в якості регуляторів гучності в аудіоапаратурі або для налаштування різних параметрів в електронних схемах.

Фоторезистор (або LDR - Light Dependent Resistor) - це резистор, опір якого змінюється залежно від інтенсивності світла. Він використовується в датчиках освітлення і різних автоматичних системах, де потрібна реакція на зміни освітленості.

Діод - це напівпровідниковий компонент, який пропускає струм тільки в одному напрямку. Він використовується для випрямлення струму, захисту схем від зворотної полярності і для різних інших цілей в електронних пристроях.

Стабілітрон - це спеціальний вид діода, який стабілізує напругу на певному рівні. Він використовується в схемах для забезпечення стабільної напруги живлення або захисту від перепадів напруги.

Конденсатор - це компонент, що зберігає електричний заряд. Він має дві обкладки, розділені ізолятором (діелектриком). Конденсатори використовуються для фільтрації сигналів, зменшення пульсацій, зберігання енергії та інших цілей.

Полярний конденсатор - це тип конденсатора, в якого є полярність: позитивний і негативний виводи. Він має більшу ємність у порівнянні з неполяризованими конденсаторами, але потребує правильної полярності підключення. Використовується в схемах постійного струму.

Амперметр - це вимірювальний прилад для вимірювання сили струму в електричному колі. Його вимірювальні одиниці - ампери (A). Амперметр підключається послідовно до кола.

Вольтметр - це вимірювальний прилад для вимірювання напруги в електричному колі. Напруга вимірюється в вольтх (В). Вольтметр підключається паралельно до компонентів кола.

NPN-транзистор - це тип біполярного транзистора, де два шари напівпровідникового матеріалу типу "n" оточені одним шаром типу "p". Він часто використовується в схемах для підсилення або переключення сигналів.

PNP-транзистор - це біполярний транзистор, в якому два шари напівпровідникового матеріалу типу "p" оточені шаром типу "n". PNP-транзистори також використовуються для підсилення і переключення сигналів, але працюють у зворотному режимі в порівнянні з NPN-транзисторами.

Резистор із зазначенням потужності - це резистор, для якого вказана максимальна потужність, яку він може витримувати без перегріву і пошкодження. Потужність вимірюється в ватах (Вт). Вибір резистора з відповідною потужністю важливий для запобігання його перегріванню і виходу з ладу в схемі.

Для подальшого вивчення радіоелектроніки, радимо перечитати книги «Р. Токхейм: Основи цифрової електроніки» та «П. Хоровиц, У. Хилл: Мистецтво схемотехніки»

Урок 2.3. ЩО ТРЕБА ЗНАТИ ПРИ НАПИСАННЯ КОДУ ДЛЯ ARDUINO.

Загалом при написання коду для Arduino ми використовуємо файл «sketch.ino», який має такий код:

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

Функція *setup()* виконує усі команди, які у себе має, при запуску Arduino (Загалом це можуть бути команди для налаштування елементів, такі як: рідкокристалічний екран; серво; двигун постійного току; тощо), а функція *loop()* Arduino виконує постійно усі команди, які у себе має.

Також треба пам'ятати про команди, які використовуються для Arduino. Ось код з базовими командами у функції *loop()* для постійного виконання команд:

```
void loop() {
    /*Вивести текстове значення у консолі.
    Можна вивести літери англійської мови*/
    Serial.println("Hello World!");

    //Таймер для очікування часу у мілісекундах
    delay(1000); // Очікувати 1 секунду

    //Назначити вивід "13" на "ВИСОКИЙ"
    digitalWrite(13, HIGH);

    //Назначити вивід "13" на "НИЗЬКИЙ"
    digitalWrite(13, LOW);
    /*Виводи у Arduino:

    Аналогові піни: A0; A1; A2; A3; A4; A5;
    Цифрові піни: 0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13;
    Живлення: 3.3V; 5V;
    Заземлення: GND;

    Аналоговим є непереривний зв'язок (Наприклад: світло; звук),
    а цифрові користуються для читання та передачі цифрового сигналу,
    такі як "0" або "1".    */

    int a = 0;
    int b = 0;
    int c = 0;
    int d = 0;

    // Умова, якщо обидві оператори порівняння дорівнюють "True"
    if (a == b && c == d) {
        ...
    } else {
        ...
    }

    // Умова, якщо хоча-би оператор порівняння дорівнює "True"
    if (a == b && c == d) {
        ...
    } else {
        ...
    }
}
```

При створенні схеми з використанням Arduino, вона на схемі позначається таким чином:

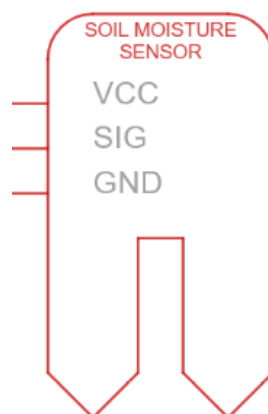


При створенні схеми з використанням Arduino, вона на схемі позначається таким чином:

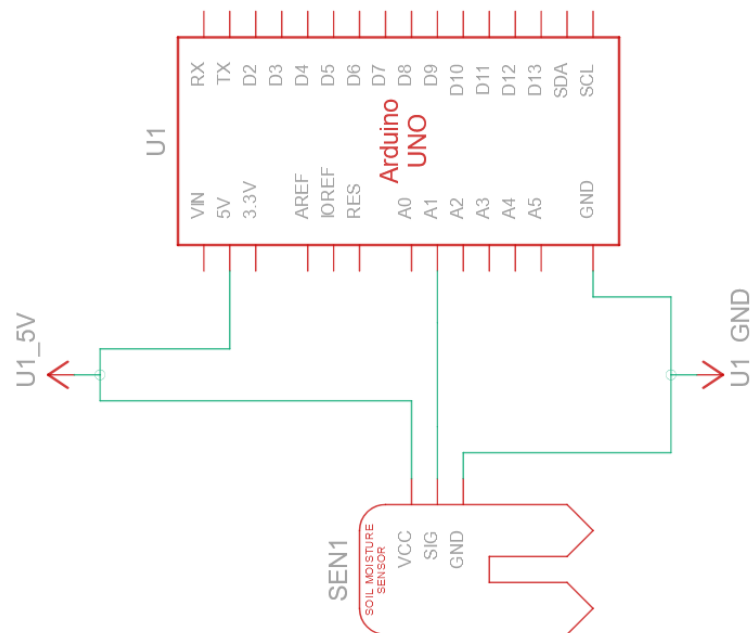
Урок 2.4. ВИКОРИСТАННЯ ДАТЧИКІВ.

Датчики – елементи радіоелектроніки, які приймають сигнал від чогось. Наприклад: датчик газу приймає сигнал завдяки газу; датчик вологості приймає сигнал завдяки води та вологості; фоторезистор (датчик наявності світла) приймає сигнал від освітлення навколишнього світу; тощо. Загалом датчики мають 3 виводи: живлення (VCC); заземлення (GND); сигнал (SIG). Є також резистори, так як: постійні резистори; потенціометри; фоторезистори; тощо, які потребують лише живлення та мають 2 виходів, де від одного йде живлення, а з іншого виходить живлення та отримуємо сигнал.

Для початку розберемось з датчиком вологості. Датчик вологості служить для отримання сигналу від води та вологості простору. Ось так він позначається у схемах:



Давайте зробимо код для Arduino, який буде відправляти у консоль інформацію про вологість. Ось його схема:



А ось його код:

```
int moisture = 0;

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

void loop()
{
  moisture = analogRead(A1); // Отримуємо результат
                             // вологості у змінній
  Serial.println(moisture);
}
```

Тепер перейдемо до датчика температури. Термістор (Датчик температури) служить для отримання інформації про температуру у навколишньому середовищі. При нагрівання навколишнього середовища збільшується напруга термістора. Є багато різновидів датчиків температури, такі як: TMP36; LM35; тощо. Але найкращим датчиком для визначення температури – TMP36, бо він може визначати як температуру менше нуля, так і температуру більше нуля, тому що він отримує дані розміру

аналогового значення. Формули для знаходження градусів Цельсіях та напруги від аналогового входу TMP36 є на початку наступної сторінки.

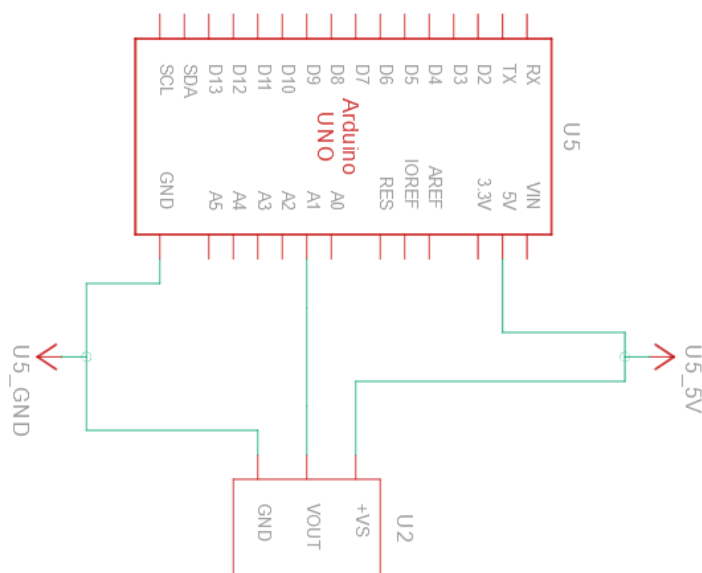
$$\text{Напруга (Вольт)} = \frac{(\text{Аналогове значення}) * 1.1}{1024}$$

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

Якщо TMP36 отримав сигнал у 50 градусів Цельсіях, то при виході він дасть 1 вольт. Ось як позначається TMP36 на схемі:



Тепер зробимо код, який буде у консолі видавати інформацію про вольт та температуру у градусах Цельсіях від термістора. Ось схема:



Ось код схеми:

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

void loop()
{
  // Продовження коду у наступній сторінці
```

```

int reading = analogRead(A1);
float voltage = (reading * 1.1) / 1024.0;
float temperatureC = (voltage - 0.5) * 100;

// Перша сторка
Serial.print(voltage);
Serial.println(" volts");

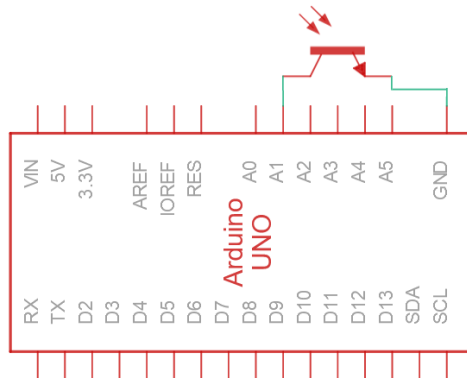
// Друга сторка
Serial.print(temperatureC);
Serial.println(" degrees C");
}

```

Також є датчики освітлення, які служать для видачі сигналу про освітлення простору. Датчик освітленості – це напівпровідниковий елемент, опору якого змінюється при зміненні світлових промінів. При наявності освітлення, він пропускає напругу. Ось як він позначається у схемах:



Тепер зробимо код, при якому датчик освітлення якщо є наявність світлових промінів, то отримуємо у консолі «1», а іншому випадку у «0». Ось схема:



Ось код:

```

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

// Продовження коду у наступній сторінці

```

```

void loop()
{
    // Отримуємо результат вологості у змінній
    int moisture = analogRead(A1);

    Serial.println(moisture);
}

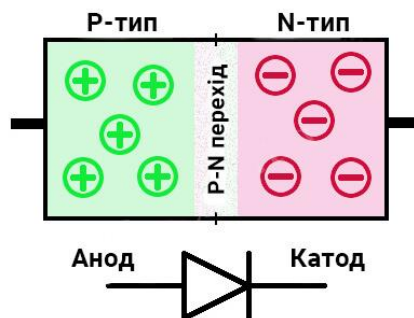
```

Загалом датчик освітленості виводить у бінарні дані, як «0» коли нема світлових промінів або «1» коли є світлові проміні.

Також є аналог датчик освітлення – фоторезистор. По властивості він теж такий, але відноситься не до датчиків, а до резисторів, такі як: потенціометр; резистор; тощо. Резистори служать для зміни сили току, а датчики служать для зміни напруги. Ось так позначається фоторезистор на схемах:



Також є аналог фоторезистора та датчику освітлення – фотодіод, який відноситься до діодів. Діод – електронний компонент, який пропускає електричний струм тільки в одну сторону – від анода до катода. Діод також називають випрямлячем, так як він перетворює змінний струм в пульсуючий постійний. Напівпровідниковий діод складається з пластинки напівпровідникового матеріалу (кремній або германій) одна сторона пластинки – з електропровідністю р-типу, тобто приймає електрони. Інша сторона віддає електрони, у неї провідність n-типу. На зовнішні поверхні пластини нанесені контактні металеві шари, до яких припаяні дротові висновки електродів діода.



Ось так позначають фотодіод на схемах:



Також є з резисторів – гнучкий сенсор, який змінює напругу при деформації цього резистора. Його схема:



Також є з резисторів – датчик схилю, який змінює напругу при зміні схилю датчика. Його схема:

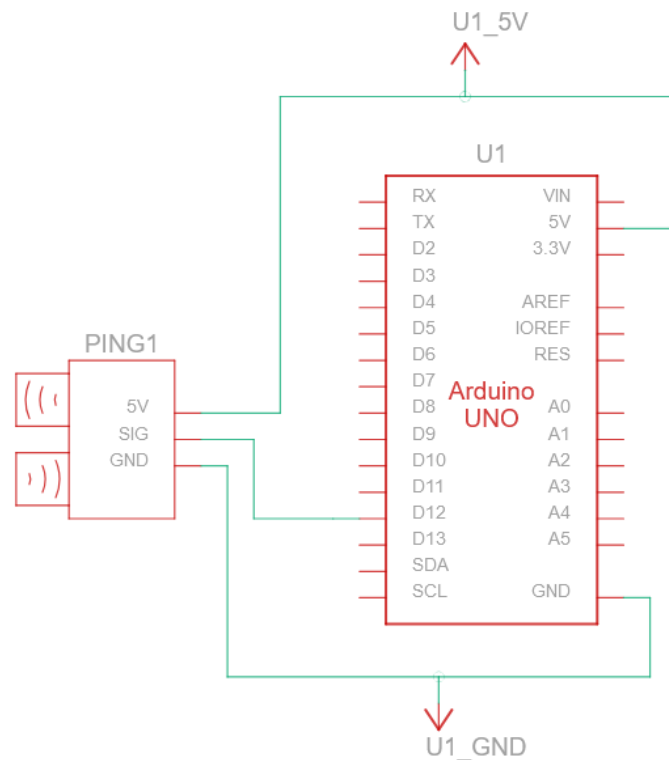


Ще є ультразвуковий датчик відстані, який відноситься до датчиків.

Ультразвуковий датчик відстані - датчик, який використовує звукові хвилі для визначення відстані до об'єкта. Він розпізнає об'єкт від 200 та менше сантиметрів. Його схема:



Тепер зробимо код для Arduino, який буде нам показувати відстань якогось об'єкта від того датчика. Ось схема:



Тепер зробимо код для цієї схеми:

```
int sig = 0;

// Ось цю функцію треба зробити
// triggerPin - пін для відправки імпульса
// echoPin - пін для прийому еха
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);

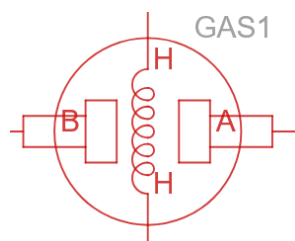
    // Повертає результат функції
    return pulseIn(echoPin, HIGH);
}

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

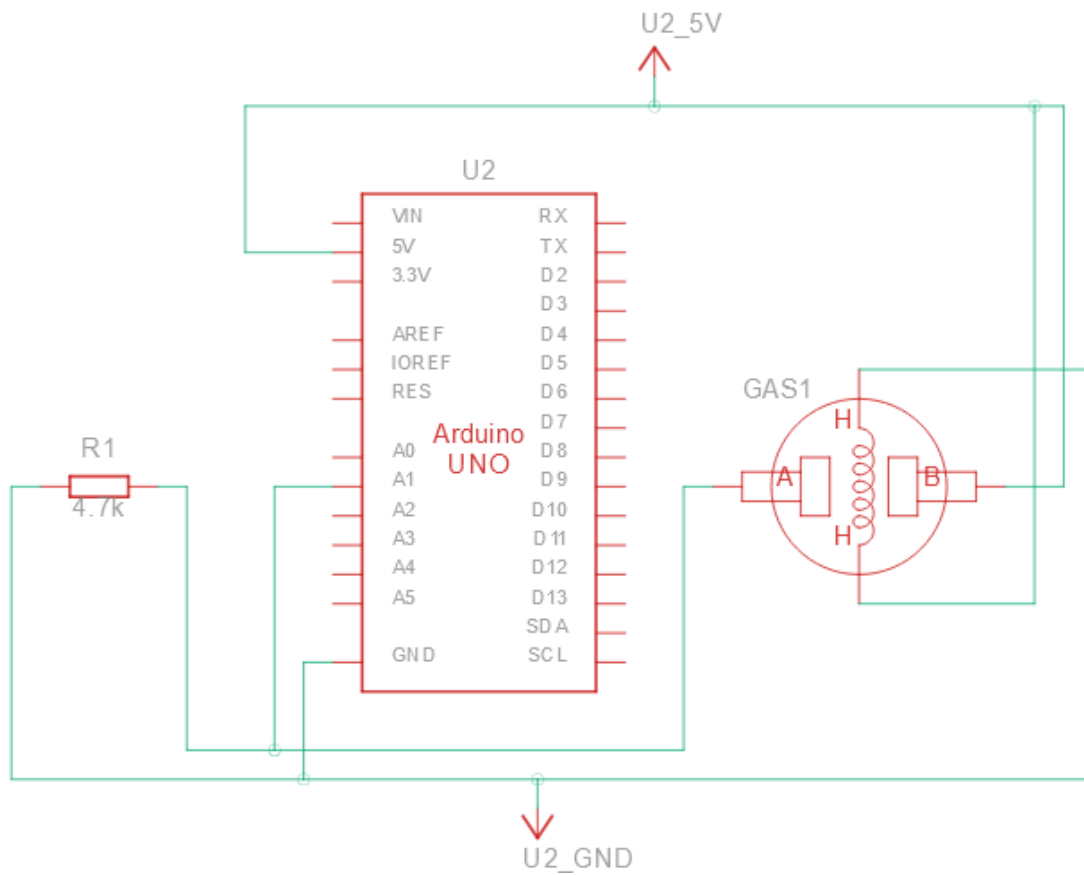
void loop()
{
    // Переведення відстані у сантиметрах
    sig = 0.01723 * readUltrasonicDistance(12, 12);

    // Показати результат у консолі
    Serial.println(sig);
}
```

Є також датчик газу, який виявляє газ, такі як: LPG (зріджені нафтові гази); ізобутан; метан; алкоголь; водень; дим; тощо. Ось як він виглядає на схемах:



Тепер зробимо код, у якому буде виводити дані від датчика. Схема:



Код:

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

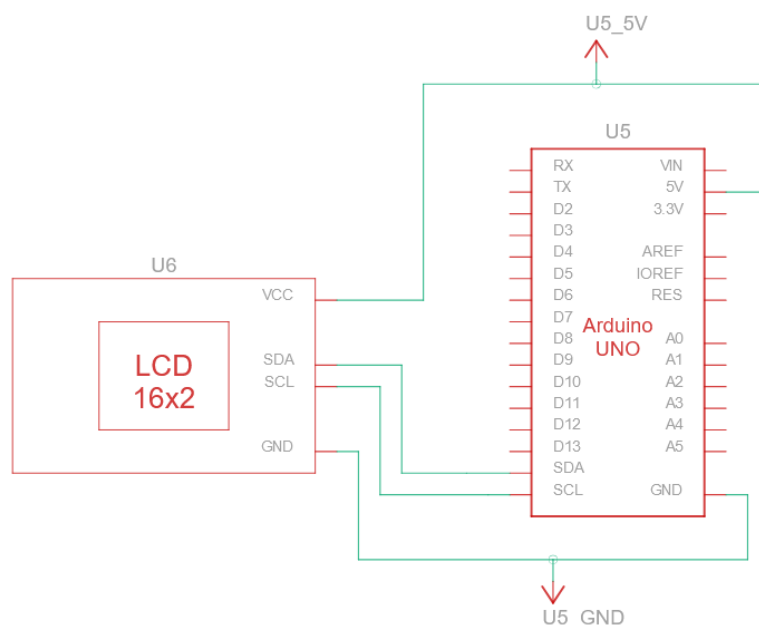
void loop()
{
  int moisture = analogRead(A1);
  Serial.println(moisture);
}
```

Урок 2.5. ВИКОРИСТАННЯ ЖИДКО-КРИСТАЛИЧНОГО ЕКРАНУ.

Жидко-кристаличний екран – елементи радіоелектроніки, які відображає текст у себе на екрані. Ось як позначається цей елемент на схемах:



Тепер створимо код для цієї схеми:



Якщо ми роздивлялись елементи, які не потребують імпорту бібліотеки, то у даному випадку при використанні жидко-кристаличного екрану треба імпортувати бібліотеку `<Adafruit_LiquidCrystal.h>`.

Ось код схеми:

```
#include <Adafruit_LiquidCrystal.h>

Adafruit_LiquidCrystal lcd_1(0);

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

void loop()
{ // Продовження коду у наступній сторінці
```

```

lcd_1.setCursor(0, 0); // Встановити координати для
                        // показу тексту
lcd_1.print("Hello World!"); // Показати текст

lcd_1.setCursor(0, 1); // Встановити координати для
                        // показу значення змінної
lcd_1.print(seconds); // Показати значення змінної

lcd_1.setBacklight(1); // Увімкнути підсвітку екрану
delay(500);
lcd_1.setBacklight(0); // Вимкнути підсвітку екрану
delay(500);
}

```

Урок 2.6. ВИКОРИСТАННЯ 7-СЕГМЕНТНОГО ЕКРАНУ.

7 сегментний екран – екран, який складається з 7 сегментів на кожну цифру. Ось як він позначається на схемах:



При сборці, 7 сегментний екран також під'єднуються однаково, як і жидкокристаличний екран. Це можна побачити схему на стр. 60. Таким чином можна теж збирати радіотехніку з 7 сегментним екраном.

```

#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();
}

```

Зверніть увагу, що на цей екран можна написати лише такі символи: «.»; « »; «0»; «1»; «2»; «3»; «4»; «5»; «6»; «7»; «8»; «9»; «-»; «_».

