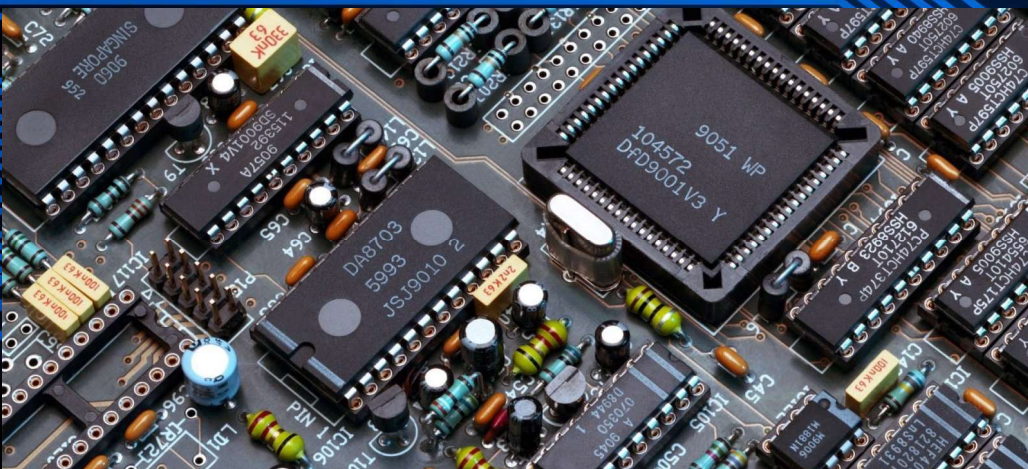


Безпека інтернет речей

Лекція №6



Лекцію проводить:
доц. Лимаренко Вячеслав Володимирович
к.т. 066-0708586

Умовні оператори та оператори вибору

У Arduino наявні логічні величини. Вони приймають два значення: правда і брехня, **true** і **false**, 1 і 0. Як тип даних по роботі з логічними величинами використовується **boolean** (синонім – **bool**), який може набувати значення 0 (**false**) або 1 (**true**). Таке саме значення повертає результат порівняння двох чисел або змінних.

Оператори порівняння:

```
== рівність (a == b);  
!= нерівність (a != b);  
>= більше або дорівнює (a >= b);  
<= менше або дорівнює (a <= b);  
> більше (a > b);  
< менше (a < b).
```

Як це працює?

Дія в дужках повертає логічне значення, що є результатом порівняння чисел. Припустимо, що $a = 10$, $b = 20$.

Дужка $(a > b)$ поверне значення **false**, тому що a менше b , тоді, як $(a != b)$ поверне **true**, т.я. a справді не дорівнює b .

Для зв'язку кількох логічних величин використовуються логічні оператори:

```
! логічне НЕ, заперечення. Є аналог - оператор not;  
&& логічне І. Є аналог - оператор and;  
|| логічне АБО. Є аналог - оператор or.
```

Операції з змінними і константами. Приклад умов

Результати операторів порівняння можна використовувати для роботи з умовами, а також для циклу **while**

```
.  
byte a = 10, b = 20;  
(a > b); // false, брехня  
(a != b); // true, правда  
  
boolean flag = true;  
flag; // true, правда  
!flag; // false, !інверсне значення  
!(a > b); // Увага! true, правда, т.я. є "!"  
  
//flagA = true, flagB = false;  
(flagA && flagB); // false, т.я. B false  
//flagA = true, flagB = true;  
(flagA and flagB); // true, т.я. обидва true  
  
//flagA = true, flagB = false;  
(flagA || flagB); // true, т.я. хоч би A true  
//flagA = false, flagB = false;  
(flagA or flagB); // false, т.я. ни A ни B не true
```


Операції з змінними і константами. Порівняння **float**

З порівнянням **float** чисел все не так просто через особливості самої моделі чисел з плаваючою точкою — обчислення іноді виконуються з невеликою похибкою, через це порівняння може працювати **неправильно!**

Приклад:

```
float val1 = 0.1;
// val1 == 0.100000000
float val2 = 1.1 - 1.0;
// val2 == 0.100000023 !!!
// здавалось би, val1 == val2
// але порівняння поверне false
if (val1 == val2); // false
```

Будьте уважні при порівнянні **float** чисел, особливо зі строгими операторами \leq , результат може бути некоректним та нелогічним!



Операції з змінними і константами. Умовний оператор **if**

Умовний оператор **if** дозволяє розгалужувати виконання програми залежно від логічних величин, тобто, результатів роботи операторів порівняння, а також безпосередньо від логічних змінних.

```
if (логічна величина) {  
    // виконується, якщо логічна величина - true  
}
```

Приклад:

```
int A = 50 ;  
int B = 100 ;  
void DigitalWrite(int pinNumber, boolean status)  
{  
    pinMode(pinNumber, OUTPUT);  
    digitalWrite(pinNumber, status);  
}  
  
void setup()  
{  
}  
  
void loop()  
{  
    if (A > B)  
    {  
        DigitalWrite(13, HIGH); // не спрацює, т.я. A < B  
    }  
}
```

Спробуйте виправити так, щоб спрацювало.

Операції з змінними і константами. Умовний оператор **else**

Оператор **else** працює в парі з оператором **if** і дозволяє передбачити дію на випадок невиконання **if**:

```
if (логічна величина) {  
    // виконується, якщо логічна величина - true  
} else {  
    // виконується, якщо логічна величина - false  
}
```

Приклад:

```
int A = 50 ;  
int B = 100 ;  
void DigitalWrite(int pinNumber, boolean status)  
{  
    pinMode(pinNumber, OUTPUT);  
    digitalWrite(pinNumber, status);  
}  
  
void setup()  
{  
}  
  
void loop()  
{  
    if (A > B)  
    {  
        DigitalWrite(13, LOW); // не буде працювати  
        DigitalWrite(12, HIGH);  
    }  
    else  
    {  
        DigitalWrite(13, HIGH); // а це буде!  
        DigitalWrite(12, HIGH);  
    }  
}
```

Операції з змінними і константами. Умовний оператор **else if**

Однією з основних функцій мікроконтролера є виконання обчислень, як із числами напряму, так і зі значеннями змінних. Найпростіші математичні дії:

```
if (логічна величина 1) {  
    // виконується, якщо логічна величина 1 - true  
} else if (логічна величина 2) {  
    // виконується, якщо логічна величина 2 - true  
} else {  
    // виконується інакше  
}
```

Приклад:

```
byte buttonState;  
if (buttonState == 1) a = 10;           // якщо buttonState 1  
else if (buttonState == 2) a = 20;     // якщо ні, але якщо buttonState 2  
else a = 30;                           // в протилежному випадку ось це
```



Зверніть увагу на даний приклад.

Використовується **else if** для вибору дії в залежності від значення однієї і тієї ж змінної. Існує оператор вибору **switch**, що дозволяє зробити код більш красивим.

Операції з змінними і константами. Порядок умов

Порядок умов відіграє дуже велику роль при оптимізації коду та спробі зробити його швидшим у деяких випадках. Суть дуже проста: логічні вирази/величини перевіряються зліва направо, і якщо хоч одне значення робить вираз невірним (брехнею), подальша перевірка умов припиняється. Наприклад якщо у виразі

```
if (a && b && c) {  
    // щось виконуємо  
}
```

хоча б `a` має значення `false`, перевірка інших виразів (`b` і `c`) не виконується. Коли це може бути важливо: наприклад, є якийсь прапор та вираз, який обчислюється прямо за умови і відразу перевіряється. У разі якщо прапор опущений, МК не буде витрачати час на зайві обчислення. Наприклад:

```
if (flag && analogRead(0) > 500) {  
    // робити що-небудь  
}
```

Якщо прапор опущений, мікроконтролер не витратиме зайві 100 мкс на роботу з АЦП, і відразу проігнорує решту логічних виразів.



Пам'ятайте, що порядок умов має значення.

Операції з змінними і константами. Тернарний оператор?

Оператор `?`, або тернарний оператор, є більш коротким аналогом для запису конструкції `if else`. Дія з оператором `?` має такий вигляд:

`умова ? вираз1 : вираз2`

Працює так: обчислюється умова, якщо вона істинна, то вся дія повертає значення виразу1, а якщо вона хибна, то вся дія повертає значення виразу2.

Приклад:

```
byte a, b;  
a = 10;  
  
// якщо a < 9, b набуває значення 200  
// інакше b набуває значення 100  
b = (a > 9) ? 100 : 200;
```

Теж саме, але «звичним» виглядом:

```
a = 10;  
if (a > 9) b = 100;  
else b = 200;
```

Аналогічним чином можна використовувати оператор `?` для виведення даних та тексту у послідовний порт

```
Serial.println((a > 9) ? "більше 9" : "менше 9");
```

Операції з змінними і константами. Тернарний оператор?

Можна зробити на операторі ? складнішу конструкцію, типу `else if`

```
void setup() {  
    Serial.begin(9600);  
    // код виводить "розмір" змінної value  
  
    byte value = 5;  
  
    // конструкція на if-else  
    if (value > 19) Serial.println("> 19");  
    else if (value > 9) Serial.println("> 9");  
    else Serial.println("< 9");  
  
    // на операторі ?  
    Serial.println(( (value > 9) ? ( (value > 19) ? "> 19" : "> 9" ) : "< 9" ));  
}
```

Операції з змінними і константами. Оператор вибору **switch**

Оператор вибору **switch** дозволяє створити зручну конструкцію, що розгалужує дії залежно від значення однієї змінної. Синтаксис:

```
switch (значення) {  
    case 0:  
        // виконати, якщо значення == 0  
        break;  
    case 1:  
        // виконати, якщо значення == 1  
        break;  
    case 2:  
    case 3:  
    case 4:  
        // виконати, якщо значення == 2, 3 або 4  
        break;  
    default:  
        // виконати, якщо значення не збігається з жодним case  
        break;  
}
```

Наявність оператора **default** не є обов'язковою.

Наявність оператора **break** є обов'язковою, інакше порівняння піде далі, як показано для case 2, 3 та 4.

Операції з змінними і константами. Оператор вибору **switch**



Потрібно бути дуже уважним під час роботи з оператором **switch**, тому що код, що знаходиться всередині фігурних дужок **switch() { }**, є одним блоком коду для всіх кейсів. Відповідно кейси **case** – лише ярлики переходу між ділянками цього блоку. ВСІ кейси знаходяться в одній області видимості, тобто всередині **switch** не можуть бути оголошені локальні змінні з однаковими іменами:

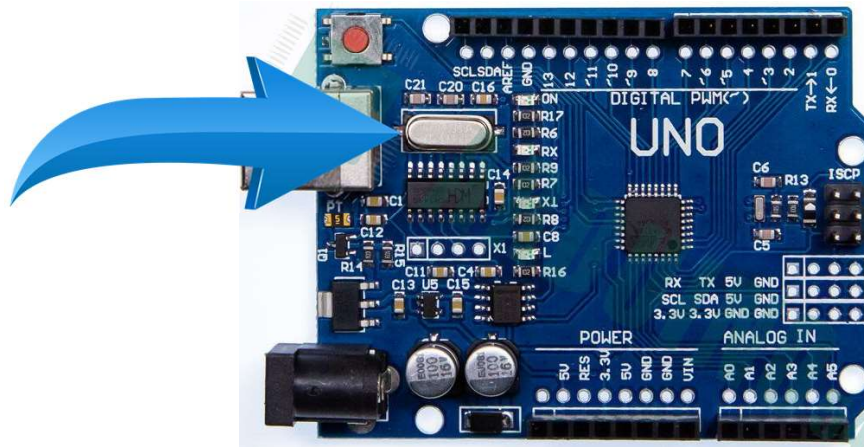
```
switch (mode) {  
    case 0:  
        long val = 100;  
        break;  
    case 1:  
        long val = 100; // призведе до помилки  
        break;  
    case 2:  
        break;  
}
```

Вкрай не рекомендується взагалі створювати локальні змінні всередині кейсів, оскільки це може зламати код!

Функції часу

Звідки взагалі мікроконтролер знає, скільки часу минає?!

Для роботи мікроконтролера життєво важливий так званий тактовий генератор, або кварцовий генератор, або ж кварц.



Кварц розташований поруч із МК на платі (також у багатьох МК є вбудований тактовий генератор), на Arduino зазвичай стоїть генератор на 16 МГц, також зустрічаються моделі на 8 МГц. Тактовий генератор виконує просту річ: він генерує імпульси на мікроконтролер зі своєю тактовою частотою. 16 МГц кварц генерує 16 мільйонів імпульсів на секунду. Мікроконтролер, «знаючи» частоту кварцу, може розрахувати час між імпульсами:

$$(16 \text{ МГц} = 0.0625 \text{ мікросекунди})$$

Функції часу

Приймають імпульси таймера так звані лічильники (Timer-counter). Це фізично розташовані всередині МК пристрої, які займаються підрахунком імпульсів тактового генератора.

Arduino має готові функції часу. В Arduino на базі ATmega328 є три лічильники, і підрахунком часу займається таймер під номером 0. Цим може займатися будь-який інший лічильник, але працюючи в Arduino IDE ви отримуєте таке налаштування, т.я. створюючи скетч в Arduino IDE, ви автоматично працюєте з бібліотекою *Arduino.h*, де і реалізовані всі функції часу.

Функції часу. Затримки

Найпростішою з точки зору використання функцією часу є затримка, їх дві:

`delay(time)` – «зупиняє» виконання коду на `time` мілісекунд. Далі функції `delay()` виконання коду не йде, крім переривань. Використовувати рекомендується тільки в крайніх випадках, або коли `delay` не впливає на швидкість роботи пристрою. `time` приймає тип даних `unsigned long` і може призупинити виконання на час від 1 мс до ~50 діб (4 294 967 295 мілісекунд) з роздільною здатністю в 1 мілісекунду. Працює на системному таймері Timer 0, тому не працює всередині переривання та при відключених перериваннях.

`delayMicroseconds(time)` – аналог `delay()`, зупиняє виконання коду на `time` мікросекунд. `time` приймає тип даних `unsigned int` і може призупинити виконання на час від 4 до 16383 мкс з роздільною здатністю 4 мкс. **Важливо:** `delayMicroseconds` працює не на таймері, як решта функцій часу в Arduino, а на рахунку тактів процесора. З цього випливає, що `delayMicroseconds` може працювати у перериванні та при відключених перериваннях.

```
void setup() {}

void loop() {
    // щось виконуємо
    delay(500); // очікуємо 0,5 с.
}
```

Затримки `delay()` краще
взагалі не використовувати
у реальному коді!



Функції часу. Функція **yield()**

Функція **delay()** блокує виконання коду. Між тим є милиця, що дозволяє виконувати інший код під час затримки. Ця милиця має назву **yield()**

```
void yield() {  
    // ваш код  
}
```

Розташований всередині неї код виконуватиметься під час роботи будь-якої затримки **delay()** у програмі! Це рішення хоч і здається безглуздим, але в той же час дозволяє швидко і без опису зайвих милиць і таймерів реалізувати пару завдань, що паралельно виконуються.

```
void setup() {  
    pinMode(13, OUTPUT);  
}  
  
void loop() {  
    digitalWrite(13, 1);  
    delay(1000);  
    digitalWrite(13, 0);  
    delay(1000);  
}  
  
void yield() {  
    // а тут можна опитувати кнопку  
    // і не пропустити натискання через delay!  
}
```

Функції часу. Функції підрахунку часу

Ці функції повертають час, що минув з моменту запуску МК. Таких функцій дві:

millis() – повертає кількість мілісекунд, що пройшли із запуску. Повертає **unsigned long**, від 1 до 4 294 967 295 мілісекунд (~50 діб), має розподільну здатність 1 мілісекунда, після переповнення скидається в 0.

Працює на системному таймері Timer 0.

micros() – повертає кількість мікросекунд, що пройшли із запуску. Повертає **unsigned long**, від 4 до 4 294 967 295 мікросекунд (~70 хвилин), має розподільну здатність у 4 мікросекунди, після переповнення скидається в 0.

Працює на системному таймері Timer 0.

Функції часу. Таймер на `millis()`

Як це працює?!

Алгоритм роботи таймеру на `millis()`:

- виконали дію;
- запам'ятали поточний час зі старту МК (в окрему змінну);
- шукаємо різницю між поточним часом та тим, що запам'ятали;
- як тільки різниця більше потрібного часу «Таймера» – виконуємо дію;
- «скидаємо» таймер;

Тут є два варіанти, прирівнювати змінну таймера з актуальним `millis()`, або збільшувати її на розмір періоду.

Функції часу. Таймер на `millis()`

Реалізація класичного «таймеру» на `millis()`

```
// змінна зберігання часу (unsigned long)
uint32_t myTimer1;

void setup() {}

void loop() {
    if (millis() - myTimer1 >= 500) {    // шукаємо різницю (500 мс)
        myTimer1 = millis();            // скидання таймера
        // виконати дію
    }
}
```

Дана конструкція «випадає» з періоду, якщо в коді є затримки та інші блокуючі ділянки, під час виконання яких `millis()` встигає збільшитися на більше ніж період. Це може бути критично, наприклад, для підрахунку часу та інших схожих ситуацій, коли період спрацьовування таймера не повинен «зміщуватися» щодо поточного часу. У той же час, якщо заблокувати виконання коду на час, більший за один період – таймер скоригує цю різницю, тому що ми його скидаємо актуальним значенням `millis()`.

Функції часу. Таймер на **millis()**

Дана конструкція жорстко відпрацьовує період, тобто не «відходить» з часом, якщо в коді присутня мала затримка, тому що час наступного спрацьовування завжди кратний періоду. Мінусом тут є те, що якщо таймер пропустить період - він «спрацює» кілька разів при наступній перевірці! Але із цієї неприємної ситуації є вихід.

```
// змінна зберігання часу (unsigned long)
uint32_t myTimer1;
int period = 500;

void setup() {}

void loop() {
    if (millis() - myTimer1 >= period) {    // шукаємо різницю (500 мс)
        myTimer1 += period;                // скидання таймера
        // виконати дію
    }
}
```

Функції часу. Таймер на `millis()`

Можна порахувати, скільки періодів потрібно «оновити» змінну таймера. У прикладі заздалегідь обчислюється час, що минув після останнього спрацьовування, і записується в змінну *timeLeft*. До змінної таймера додаємо період, помножений на кількість переповнень. Якщо виклик таймера не був пропущений – то множити будемо на 1. Цілочисленний поділ (*timeLeft/period*) дозволяє отримати цілу кількість переповнень, тому дужки стоять саме так. Дана конструкція таймера дозволяє жорстко дотримуватися періоду виконання і не боїться пропущених викликів, але вимагає кількох додаткових обчислень, у тому числі «повільне» ділення, не забувайте про це.

```
uint32_t myTimer1; // uint32_t це друга назва типу даних unsigned long
int period = 500; // період 0,5 с
void setup() {}
void loop() {
    uint32_t timeLeft = millis() - myTimer1;
    if (timeLeft >= period) {
        myTimer1 += period * (timeLeft / period);
    }
}
```

Функція `millis()` повертає тип даних `unsigned long`. Якщо зробити змінну типу `int`, вона переповниться через 32.7 секунди. Але `millis()` теж обмежений числом 4294967295, і при переповненні теж скинеться в 0. Зробить він це через $4294967295/1000/60/60/24 = 49.7$ діб.

Функції часу. Таймер на **millis()**

Багатозадачність: хочемо виконувати одну дію двічі на секунду, другу – тричі, і третю – 10. Знадобиться 3 змінні таймери та 3 конструкції:

```
// Змінна зберігання часу (unsigned long)
uint32_t myTimer1, myTimer2, myTimer3;

void setup() {}

void loop() {
    if (millis() - myTimer1 >= 500) {    // таймер на 500 мс (2 рази в с.)
        myTimer1 = millis();             // скидання таймеру
        // виконати дію 1
        // 2 рази на секунду
    }
    if (millis() - myTimer2 >= 333) {    // таймер на 333 мс (3 рази в с.)
        myTimer2 = millis();             // скидання таймеру
        // виконати дію 2
        // 3 рази на секунду
    }
    if (millis() - myTimer3 >= 100) {    // таймер на 100 мс (10 раз в с.)
        myTimer3 = millis();             // скидання таймеру
        // виконати дію 3
        // 10 рази на секунду
    }
}
```

Функції часу. Таймер на `millis()`. Що робити не треба

В інтернеті часто можна зустріти таку конструкцію: умова виконується, коли залишок від ділення `millis()` на період дорівнює «0».

```
if (millis() % period == 0) {  
    // якась дія  
}
```

Використовувати цей варіант **не можна** з цілого ряду причин:

- операція «залишок від ділення» досить важка і повільна, розміщення кількох таких «таймерів» в основному циклі програми сильно збільшить час її виконання та уповільнить роботу всієї програми загалом;
- у реальній програмі може створюватися затримка тривалістю довше 1 мс (наприклад, виведення на матрицю адресних світлодіодів займає ~30 мс) і існує досить високий ризик пропуску такого «таймера»;
- у той же час умова спрацьовування таймера буде правильною цілу мілісекунду і дія по таймеру може виконатися кілька разів поспіль, що неприпустимо в більшості випадків;

Висновок: **у жодному разі не використовуйте цю конструкцію**

Масиви. Оголошення масиву

Масив (array) – це сукупність змінних одного типу, до яких звертаються з допомогою загального імені та індексу, тобто, номеру елемента в масиві. По суті, це набір змінних, які називаються одним ім'ям і мають особисті номери. Для оголошення масиву достатньо вказати квадратні дужки після імені змінної, тип даних – будь-який.

Вказати компілятору розмір масиву можна двома способами: явним числом у квадратних дужках, або при оголошенні відразу присвоїти в кожному комірку значення, тоді компілятор сам порахує їх кількість.



```
sketch_mar08a $  
1 15 145 26 33  
9 18 15 5 11  
1 185 12 15 237
```

The screenshot shows the Arduino IDE interface with the serial monitor open. The title bar indicates 'sketch_mar08a | Arduino 1.8.16'. The menu bar includes 'Файл', 'Правка', 'Скетч', 'Інструменти', and 'Допомога'. The toolbar contains icons for checking, running, uploading, and downloading. The serial monitor shows the output of a program, displaying a 3x5 grid of numbers. The status bar at the bottom shows 'F5 S), 240MHz (WiFi/BT), QIO, 80MHz, 4MB (32Mb), 921600, None на COM4'.

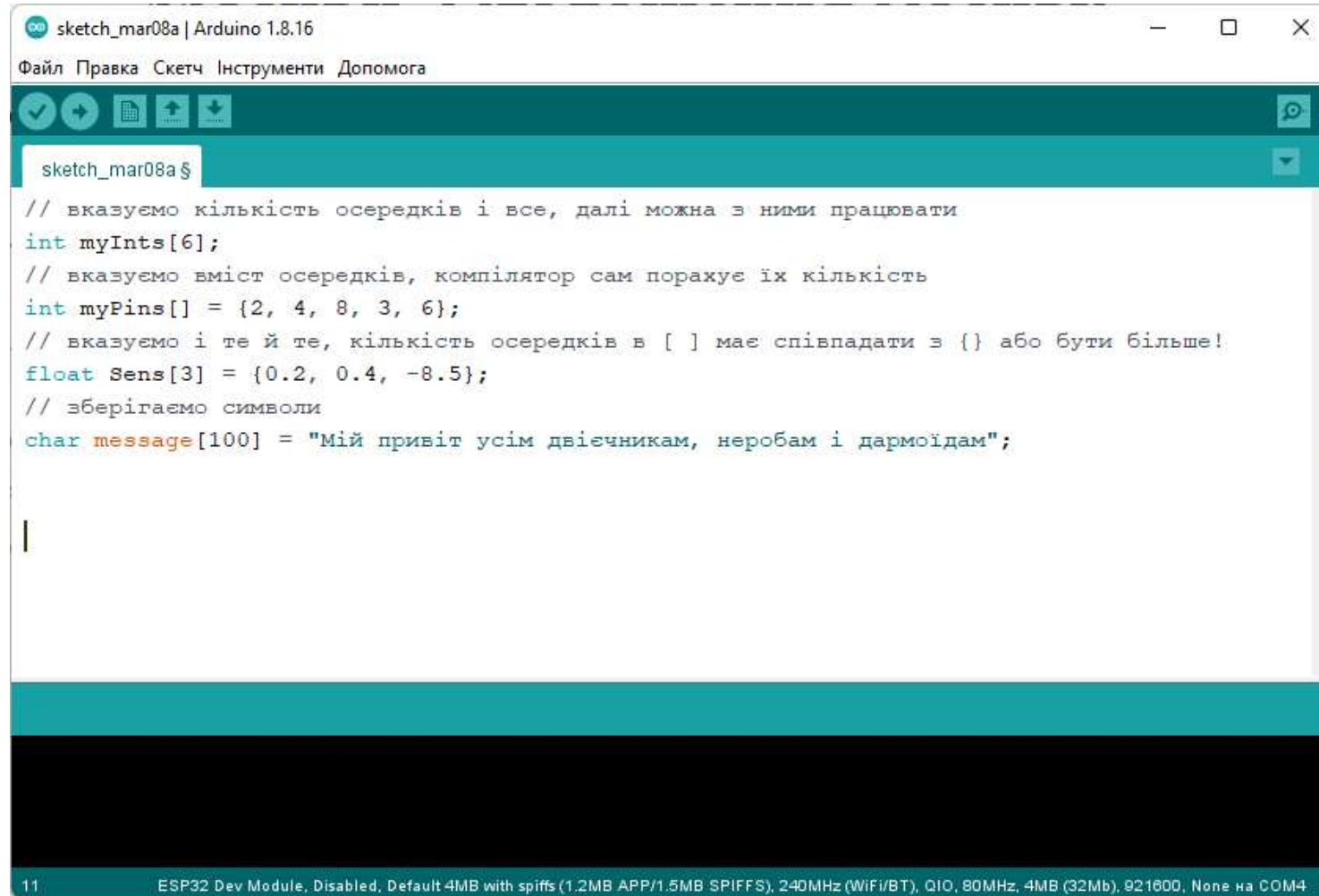


Приклад використання: масив значень електронних ключів iButton

масив міток часу відкривання/закривання приміщення...

Масиви. Оголошення масиву.

Приклад оголошення масиву різними способами




```
sketch_mar08a | Arduino 1.8.16
Файл Правка Скетч Інструменти Допомога

sketch_mar08a$
// вказуємо кількість осередків і все, далі можна з ними працювати
int myInts[6];
// вказуємо вміст осередків, компілятор сам порахує їх кількість
int myPins[] = {2, 4, 8, 3, 6};
// вказуємо і те й те, кількість осередків в [ ] має співпадати з {} або бути більше!
float Sens[3] = {0.2, 0.4, -8.5};
// зберігаємо символи
char message[100] = "Мій привіт усім двічникам, неробам і дармоїдам";

11 ESP32 Dev Module, Disabled, Default 4MB with spiffs (1.2MB APP/1.5MB SPIFFS), 240MHz (WiFi/BT), QIO, 80MHz, 4MB (32Mb), 921600, None на COM4
```

Масиви. Оголошення масиву.

 Важливий момент: розмір **глобального масиву** має бути заданий константою, тобто конкретним числом, константою **const** або константою **#define**. Справа в тому, що розмір глобального масиву повинен бути відомий до початку виконання програми, адже він виділяється з пам'яті і поруч із ним розташовані інші глобальні змінні.

```
// розміри
#define arr1_size 10
const byte arr2_size = 20;
byte arr3_size = 30;

// масиви
int arr0[5];
int arr1[arr1_size];
int arr2[arr2_size];
//int arr3[arr3_size];    // призведе до помилки! Поміркуйте, чому?!
```

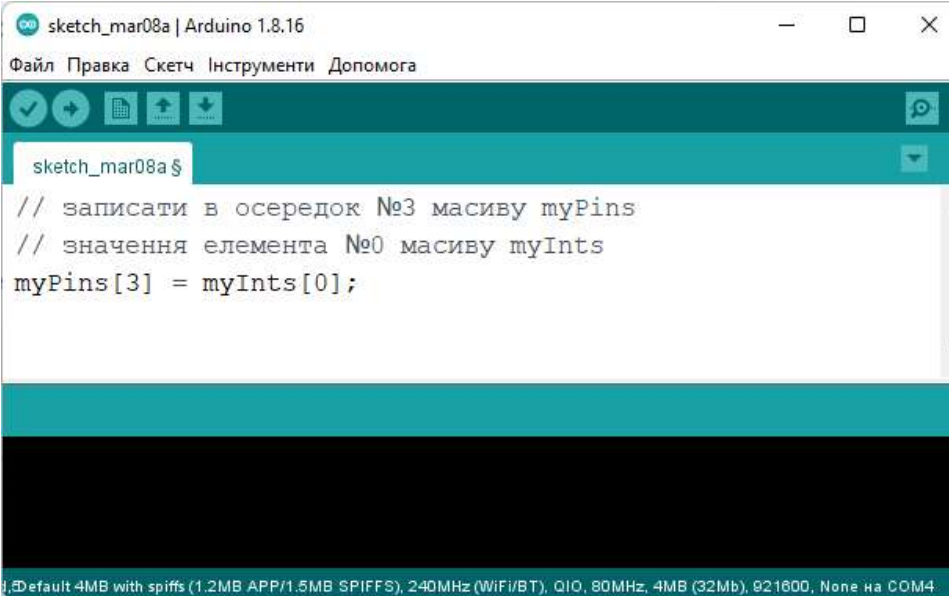
Розмір **локального масиву** (створеного всередині функції) може бути заданий змінною, тому що локальний масив виділяється з динамічної пам'яті в процесі роботи програми, і може бути будь-якого розміру:

```
void setup() {
    byte localArr_size = 10;
    int arrLocal[localArr_size];
}
```

Масиви. Звернення до елементів

Звернення до елемента масиву здійснюється точно так саме, у квадратних дужках.

Важливо пам'ятати, що відлік у програмуванні починається з «0», і перший елемент масиву має номер 0 (нуль)

A screenshot of the Arduino IDE interface. The title bar shows 'sketch_mar08a | Arduino 1.8.16'. The menu bar includes 'Файл', 'Правка', 'Скетч', 'Інструменти', and 'Допомога'. The toolbar contains icons for opening, saving, compiling, and uploading. The main text area shows a C++ sketch with two comments in Ukrainian: '// записати в осередок №3 масиву myPins' and '// значення елемента №0 масиву myInts', followed by the code line 'myPins[3] = myInts[0];'. The status bar at the bottom displays hardware specifications: 'Default 4MB with spiiffs (1.2MB APP/1.5MB SPIFFS), 240MHz (WiFi/BT), QIO, 80MHz, 4MB (32Mb), 921600, None на COM4'.

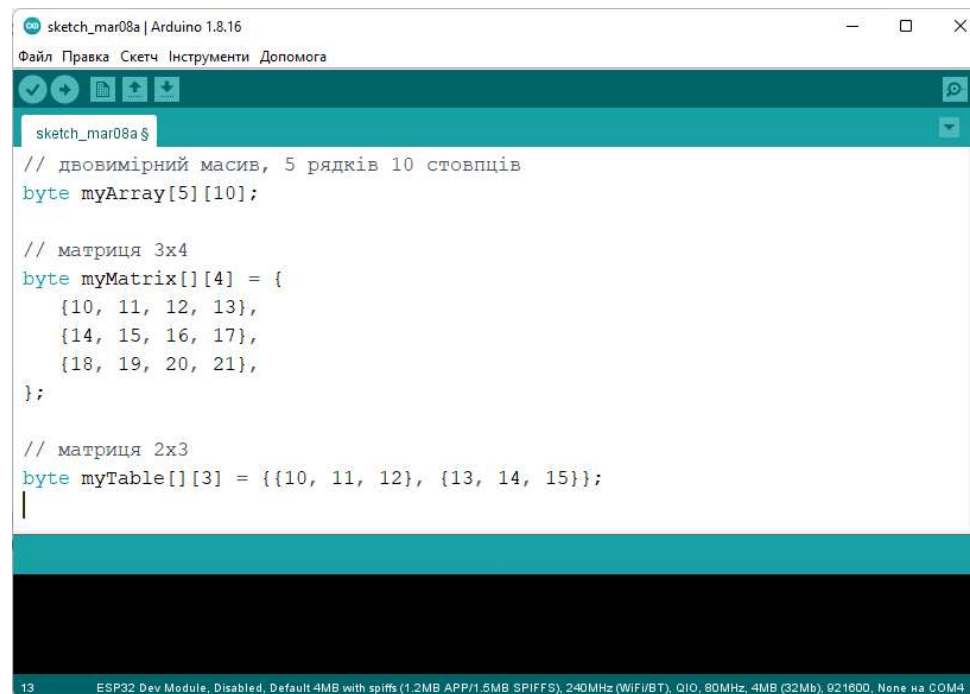
```
sketch_mar08a $  
// записати в осередок №3 масиву myPins  
// значення елемента №0 масиву myInts  
myPins[3] = myInts[0];
```

Масиви. Багатовимірні масиви

Ми розглядали **одновимірні масиви**. У одновимірному масиві елементи визначаються просто порядковим номером.

Можна ініціювати й **багатовимірні масиви**, у яких елемент матиме кілька номерів. Наприклад двовірний масив, він же матриця, він же таблиця, кожен елемент має номер рядка і стовпця.

Задається такий масив так: *тип ім'я[рядків][стовпців]*

A screenshot of the Arduino IDE interface. The title bar reads 'sketch_mar08a | Arduino 1.8.16'. The menu bar includes 'Файл', 'Правка', 'Скетч', 'Інструменти', and 'Допомога'. The toolbar contains icons for opening, saving, and running. The code editor shows the following C++ code:

```
sketch_mar08a $  
  
// двовимірний масив, 5 рядків 10 стовпців  
byte myArray[5][10];  
  
// матриця 3x4  
byte myMatrix[][4] = {  
    {10, 11, 12, 13},  
    {14, 15, 16, 17},  
    {18, 19, 20, 21},  
};  
  
// матриця 2x3  
byte myTable[][3] = {{10, 11, 12}, {13, 14, 15}};  
|
```

The status bar at the bottom displays: '13 ESP32 Dev Module, Disabled, Default 4MB with spiffs (1.2MB APP/1.5MB SPIFFS), 240MHz (WiFi/BT), QIO, 80MHz, 4MB (32Mb), 921600, None на COM4'.

Приклади оголошення багатовимірних масивів

Масиви. Багатовимірні масиви



Дуже важливо пам'ятати, що при оголошенні масиву з вручну вписаними даними потрібно обов'язково вказати розмір кількості осередків у вимірі на 1 менше розмірності масиву (для двовимірного обов'язково вказати розмір одного з вимірювань, для тривимірного два і т.д.).

Після останнього члена масиву можна ставити кому, це не призведе до помилки.

Звернення до елемента у багатовимірному масиві:

```
// матриця 3x4
byte myMatrix[][4] = {
    {10, 11, 12, 13},
    {14, 15, 16, 17},
    {18, 19, 20, 21},
};

// міняємо 12 на 20, осередок 0,2
myMatrix[0][2] = 20;
```

Дуже корисним є масив рядків (масивів літер), що дозволяє впорядковано зберігати назви пунктів меню або інших подібних речей. Такий масив має бути оголошений за допомогою адресного оператора * (зірочка):

```
const char *names[] = {
    "Студенти",    // 0
    "Відмінники",  // 1
    "Ледарі",      // 2
};
```

Масиви. Багатовимірні масиви

```
const char *names[] = {  
    "Студенти",    // 0  
    "Відмінники",  // 1  
    "Ледарі",      // 2  
};
```

Звернення до *names[2]* допоможе вивести слово «Ледарі» на монітор порту або на дисплей

```
// вивести в порт слово "Ледарі"  
Serial.println(names[2]);
```



- З елементами масивів можна робити такі ж дії, як із звичайними змінними
- Масивом може бути майже будь-який тип даних: цілочисленні, дробові, масив структур ...
- Область видимості так само застосовується до масивів, адже масив - це звичайна змінна.

Масиви. Масив символів

Символ є змінною (або константою) типу **char** і зберігає код літери в таблиці символів. Створений для зручності програміста, щоб він міг працювати не кодами, а з символами, що читаються. Символи з'єднуються у слова, вони називаються *рядки*. Є два набори інструментів для роботи з символами: звичайні рядки (масиви символів) та String-рядки.

Масив символів (char array) – це просто масив даних типу **char**. Основні особливості: максимальний розмір масиву рядка має бути відомий заздалегідь, і до кожного елемента такого рядка можна звернутися квадратними дужками. Будь-який текст, явно ув'язнений у "*подвійні лапки*", сприймається програмою як масив символів. Будучи звичайним масивом, рядок є вказівником на перший елемент (тобто програма конкретно знає лише те, де рядок починається). Вбудовані функції роботи з рядками орієнтуються на нульовий символ, який обов'язково знаходиться в кінці рядка. Таким чином і визначається довжина рядка: від початку до нульового символу.

Основною відмінністю **String-рядка** від масиву символів є те, що рядок – динамічний масив, у якого не потрібно вказувати розмір, він може змінюватись у процесі роботи програми. Також рядок є не просто типом даних, а об'єктом дуже потужного класу однойменної бібліотеки String, яка автоматично підключається до коду та додає зручні інструменти для роботи з текстом: поділ, обрізка, пошук та заміна тощо. Рядок може бути створено з будь-якого типу даних і перетворено майже в усі з них.

Масиви. Масив символів. String-рядки

Приклад, з якого буде зрозуміло, як оголосити рядок і як з ним працювати, а також враховані деякі тонкощі:

```
String string0 = "Привіт слухачі"; // Заповнюємо словами в лапках
String string1 = String("lol") + String("kek"); // сума двох рядків
String string2 = String('a'); // рядок із символу в одинарних лапках
String string3 = String("Це string"); // конвертуємо рядок у String
String string4 = String(string3 + "тут щось є"); // Складаємо рядок string3 з текстом у лапках
String string5 = String(13); // конвертуємо з числа в String
String string6 = String(20, DEC); // конвертуємо з числа із зазначенням базису (десятковий)
String string7 = String(45, HEX); // конвертуємо з числа із зазначенням базису (16-ковий)
String string8 = String(255, BIN); // конвертуємо з числа із зазначенням базису (двійковий)
String string9 = String(5.698, 3); // з float із зазначенням кількості знаків після коми (тут 3)

// Рядки можна складати один з одним
String string10 = string0 + string1; // string10 дорівнює Привіт слухачіlol kek

// можна формувати назву зі шматочків, наприклад для роботи з файлами. Навіть із дефайнів
define NAME "speed"
define TYPE "-log"
define EXT ".txt"

// при додаванні достатньо вказати String 1 раз для першого рядка
String filename = String(NAME) + TYPE + EXT; // filename дорівнюватиме speed-log.txt

// Доступ до елемента рядка працює за таким же механізмом, як масив
string0[0] = 'a'; // одинарні лапки, т.я. привласнюємо ОДИННИЙ СИМВОЛ!
// тепер замість "Привіт слухачі" у нас "аривіт слухачі"
```

Масиви. Масив символів. String-рядки

Рядки можна оголошувати великою кількістю способів, а також складати рядки, як числа оператором +.

Рядки є об'єктами класу String, і цей клас має величезну кількість зручних методів роботи з рядками

Запам'ятайте: рядки – дуже важкий інструмент, дуже повільний і займає багато пам'яті: просто наявність рядків (від одного символу і більше) в прошивці займає +5% Flash пам'яті, т.я. підключається сам «інструмент» – клас String. Для невеликих проектів це не страшно, пам'яті завжди буде достатньо.

Також неакуратне використання рядків може призводити до фрагментації оперативної пам'яті та зависання програми.

Крім набору методів бібліотека String має кілька операторів з якими можемо:

- працювати з елементами String-рядків як із масивами: *myString[2] = 'a';*
- порівнювати String-рядки між собою: *if (myString1 == myString2)*
- порівнювати String-рядки з масивами символів: *if (myString1 == "kek")*
- ініціалізувати String-рядки будь-яким чисельним типом даних, символом, масивом символів та масивом символів усередині макросу *F()*: *String myString = 10.0;*
- додати до рядка будь-який чисельний тип даних, символ або масив символів: *myString += 12345;*
- «збирати» рядки додаванням з будь-яких типів даних. Якщо перший (лівий) доданок не є String – потрібно перетворити на (String). Інші «підтягнуться» самі: *String str = (String)10 + "value" + var + ',' + 3.14;*

Масиви. Масив символів. String-рядки. Довжина рядка

Коментар з приводу довжини рядка: на відміну від `char array`, дізнатися довжину String-рядка можна тільки за допомогою методу `length()` (бо String-рядок є динамічним масивом, а `sizeof()` виконується на етапі компіляції):

```
String textString = "Слово";  
sizeof(textString); // поверне 6 ПРИ БУДЬ-ЯКІЙ ДОВЖИНІ РЯДКУ  
textString.length(); // Поверне 5
```

Масиви. Масив символів. String-рядки. Запобіжні заходи

Рядок по суті є **динамічним масивом**, його розмір може змінюватись по ходу програми. Це дуже зручно, але й дуже небезпечно, тому що пам'ять може фрагментуватись, закінчитися тощо. Розглянемо кілька базових принципів безпечної роботи з рядками:

❑ Якщо потрібно передати String-рядок у функцію – робіть це за посиланням. Це позбавить програму дублювання шматка даних, адже через досить великий рядок оперативна пам'ять може закінчитися і програма зависне. Приклад: `void someFunc(String &str);` – функція приймає посилання на рядок. На використанні функції це ніяк не позначиться, але при виклику не буде створюватися копія рядка.

❑ Не викликайте зайвий раз перетворення в String, бібліотека зробить це за вас. Наприклад, не потрібно писати `myString += String(value);`, досить просто `myString += value;`, при створенні довгого рядка шляхом додавання нових даних по шматочку – це врятує від фрагментації пам'яті.

❑ Обгортайте ділянки коду з об'ємною роботою з рядками в {фігурні дужки}: локально створені String-рядки будуть видалені з пам'яті відразу після }, що може запобігти фрагментації та переповненню пам'яті.

Масиви. Масив символів

Масиви символів, вони ж «**char array**» є ще одним способом роботи з текстовими даними. Цей варіант має набагато менше можливостей по роботі з текстом, зате займає менше місця в пам'яті (не використовується бібліотека String) і працює значно швидше. До масиву символів застосовуються ті самі правила, які працюють для звичайних масивів.

Приклад (оголосимо масиви символів у різний спосіб)

```
// оголосити масив та задати текст символами
// Розмір буде врахований компілятором
char helloArray[] = {'H', 'e', 'l', 'l', 'o'};

// Але строкові масиви можна оголошувати і так:
char helloArray2[] = "Hello!";

// Можна оголосити масив більшого розміру, ніж початковий текст.
// Буде вільне місце під інший текст у майбутньому
char helloArray3[100] = "Hello!";

// Ось так можна оголосити довгий рядок
char longArray[] = "Швидка бура лисиця "
                  "перестрибує через ледачого собаку";
```

Масиви. Масив символів

Можна працювати з елементами рядків як з масивами:

```
helloArray2[0] = 'L';           // замінимо елемент  
// тепер helloArray2 == "Lello!"
```

На відміну від рядків, масиви символів **не можна**:

```
helloArray3 += textArray; // складати  
textArray = "new text"; // Привласнювати РЯДОК після ініціалізації  
if (helloArray == helloArray2); // порівнювати
```

Важливий момент: будь-який “текст у лапках” у коді програми є масивом символів, а точніше `const char*` : вказівник, оскільки це масив, і константа – тому що текст введено до компіляції і під час роботи програми не може змінитися. Масив символів може бути перетворений на String-рядок, але сам по собі відношення до String не має!

При ініціалізації масиву символів "текстом у лапках" створюється масив з розміром на 1 більше, ніж кількість символів у тексті: компілятор дописує в кінець рядка нульовий символ NULL, завдяки якому різні інструменти роботи з рядками будуть бачити довжину рядка: від першого символу і до NULL.

Масиви. Масив символів. Довжина рядка char array

Для визначення довжини тексту можна використовувати оператор `strlen()`, який повертає кількість символів у масиві. Порівняємо його роботу з оператором `sizeof()`:

```
char textArray[100] = "World";  
sizeof(textArray); // Поверне 100  
strlen(textArray); // Поверне 5
```

Тут оператор `sizeof()` повернув кількість байт, які займає масив. Масив спеціально оголошено з розміром більшим, ніж текст, що міститься в ньому. А ось оператор `strlen()` порахував та повернув кількість символів, які йдуть з початку масиву і до нульового символу наприкінці тексту (без його обліку). А ось такий буде результат при ініціалізації без зазначення розміру масиву:

```
char text[] = "Hello";  
strlen (text); // Поверне 5 ( "зчитаних" символів)  
sizeof(text); // Поверне 6 (байт)
```

Масиви. Масив рядків

Дуже потужною фішкою масивів символів є можливість створити один масив із кількома рядками, та звертатися до них за номером. Виглядає це так:

```
// Оголошуємо масив рядків
const char *names[] = {
    "Period", // 0
    "Work",   // 1
    "Stop",   // 2
};

// Виводимо третій елемент
Serial.println(names[2]); // виведе Stop
```

Цей спосіб роботи з рядками хороший тим, що рядки зберігаються під номерами, і це дуже зручно при роботі з дисплеями і, зокрема, при створенні текстового меню: практично всі бібліотеки дисплеїв вміють виводити масив символів однією командою.

Масиви. Масив рядків. F() macro

Рядки (масиви символів) є дуже важким елементом, адже текст у рядку зберігається в оперативній пам'яті мікроконтролера, а її не так вже й багато. Існує готовий інструмент, що дозволяє зручно зберігати текстові дані у Flash пам'яті мікроконтролера. Цей спосіб хороший для виведення фіксованих текстових даних, наприклад, у монітор порту або на дисплей:

```
Serial.println(F("Hello, World!"));
```

Рядок «Hello, World!» буде записано у Flash пам'ять і не займе 14 байт (13 + нульовий) в оперативній.

Масиви. Масив рядків. Економія пам'яті

«Рядки» в масиві рядків теж зберігаються в оперативній пам'яті, а **F() macro** до них не можна.

Такий код приведе до помилки:

```
const char *names[] = {  
    F("Period"),    // 0  
    F("Work"),      // 1  
    F("Stop"),      // 2  
};
```

Масив рядків можна зберегти в PROGMEM, програмній пам'яті мікроконтролера, Flash. Ось таку конструкцію можна використовувати як шаблон:

Складно та громіздко, але при великому обсязі текстових даних це може врятувати проект. Наприклад, під час створення пристрою з текстовим меню на дисплеї.

```
// оголошуємо наші "рядки"  
const char array_1[] PROGMEM = "Period";  
const char array_2[] PROGMEM = "Work";  
const char array_3[] PROGMEM = "Stop";  
  
// Оголошуємо таблицю посилань  
const char* const names[] PROGMEM = {  
    array_1, array_2, array_3,  
};  
  
void setup() {  
    Serial.begin(9600);  
  
    char arrayBuf[10]; // створюємо буфер  
  
    // Копіюємо в arrayBuf за допомогою вбудованого strcpy_P  
    strcpy_P(arrayBuf, (char*)pgm_read_word(&(names[1])));  
  
    Serial.println(arrayBuf); // виведе Work  
}
```



Цикли

Основний цикл будь-якої програми для Arduino – `loop()`.

Цикл це рамки, код усередині яких виконується зверху донизу і повторюється з початку, коли досягає кінця. Продовжується це доти, доки виконується якась умова. Є два основних цикли, з якими працюватимемо, це `for` і `while`.

The screenshot shows the Arduino IDE window titled "sketch_mar08b | Arduino 1.8.16". The menu bar includes "Файл", "Правка", "Скетч", "Інструменти", and "Допомога". The toolbar contains icons for checking, running, saving, and uploading. The sketch editor shows the following code:

```
sketch_mar08b $  
  
void loop() {  
  // put your main code here, to run repeatedly:  
  
}
```

Below the editor is a teal bar with a search icon. At the bottom is the serial monitor, which displays "Arduino Version: 1.8.16". The status bar at the very bottom shows hardware details: "with spiiffs (1.2MB APP/1.5MB SPIFFS), 240MHz (WiFi/BT), QIO, 80MHz, 4MB (32Mb), 921600, None на COM4".

Цикли. Цикл for

Цикл **for**, або **лічильник**, у різних варіаціях цей цикл є і в інших мовах програмування, але на C++ він має дуже гнучке налаштування.

При створенні цикл приймає три «значення/налаштування»:

- ініціалізація;
- умова;
- зміна.

Цикл **for** зазвичай містить змінну, яка змінюється протягом роботи циклу, ми можемо користуватися її значенням, що змінюється, у своїх цілях. Змінна локальна для циклу, якщо вона створюється при ініціалізації.

Ініціалізація – тут зазвичай надають початкове значення змінній циклу.

Наприклад: `int i = 0;`

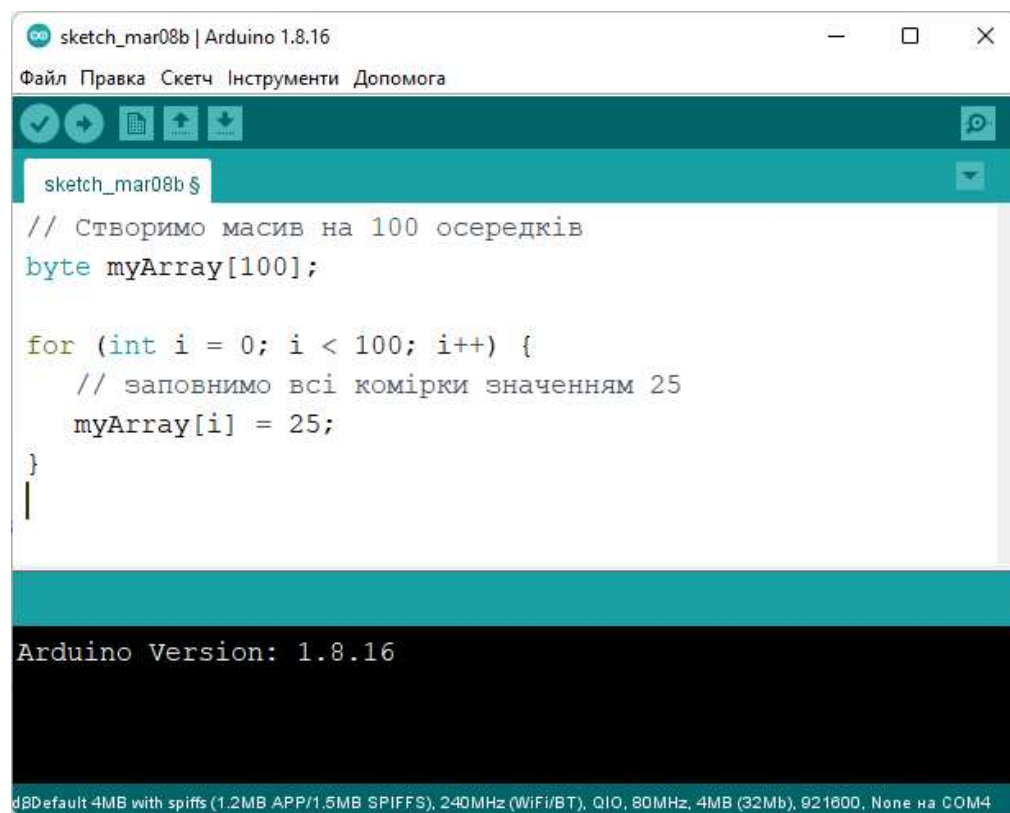
Умова – тут задається умова, при якій виконується цикл. Як тільки умова порушується, цикл завершує роботу. Наприклад: `i < 100;`

Зміна – тут вказується зміна змінної циклу кожної ітерації. Наприклад: `i++;`



Цикли. Цикл for

Приклад: у тілі циклу можна користуватися значенням змінної i , яка прийме значення від 0 до 99 протягом циклу, після цього цикл завершується. Використаємо це для автоматичного заповнення циклу



```
sketch_mar08b | Arduino 1.8.16
Файл Правка Скетч Інструменти Допомога

sketch_mar08b $
// Створимо масив на 100 осередків
byte myArray[100];

for (int i = 0; i < 100; i++) {
    // заповнимо всі комірки значенням 25
    myArray[i] = 25;
}

Arduino Version: 1.8.16

Default 4MB with spiffs (1.2MB APP/1.5MB SPIFFS), 240MHz (WiFi/BT), QIO, 80MHz, 4MB (32Mb), 921600, None на COM4
```

Цикли. Цикл for

Саме за допомогою циклу **for** дуже часто працюють із масивами. Можна, наприклад, скласти всі елементи масиву для пошуку середнього арифметичного

```
// Створимо масив даних
byte myVals[] = {5, 60, 214, 36, 98, 156};

// створюємо змінну для зберігання суми
// обов'язково ініціалізуємо її "0"
int sum = 0;

for (int i = 0; i < 6; i++) {
    // підсумовуємо весь масив у sum
    sum += myVals[i];
}

// розділимо sum на кількість елементів
// і отримаємо середнє арифметичне!
sum /= 6;

// Вийшло 94!
```

Цикли. Цикл for

Що ж до особливостей використання `for` у мові C++: будь-яке його налаштування є обов'язковим, тобто його можна вказувати для якихось особливих алгоритмів.

Не забувайте, що роздільники налаштувань (точка з комою) обов'язково повинні бути присутніми на своїх місцях, навіть якщо налаштувань немає!

Приклад

```
// є якась змінна
int index = 0;

for(; index < 60; index += 10) {
    // змінна index набуває значень
    // 0, 10, 20, 30, 40, 50
}
```

У циклі `for` можна зробити кілька лічильників, кілька умов і кілька інкрементів, розділяючи їх за допомогою оператора «,». Приклад

```
// оголошуємо i та j
// додаємо i+1 та j+2
for (byte i = 0, j = 0; i < 10; i ++, j + = 2) {
    // Тут i змінюється від 0 до 9
    // i j змінюється від 0 до 18
}
```

Цикли. Цикл for

У циклі може взагалі не бути налаштувань, і такий цикл можна вважати вічним, замкнутим:

```
for(;;) {  
    // Виконується вічно...  
}
```

Використання замкнутих циклів не дуже вітається, але іноді є дуже зручним способом зловити якесь значення, або дати програмі зависнути при настанні помилки. З такого циклу можна вийти за допомогою оператора **break**.

Цикли. Цикл for each

У компіляторі з'явилася підтримка аналога `foreach`, який є в деяких інших мовах програмування. Реалізація дозволяє скоротити код для проходження за будь-яким масивом даних.

Типовий приклад виведення масиву чисел на порт:

```
int vals[] = {10, 11, 12, 13, 14};
for (int i = 0; i < sizeof(vals); i++) {
    Serial.println(vals[i]);
}
```

ми завели цикл `for` зі змінною-лічильником `i`, яка змінюється від 0 до розміру масиву, який обчислюємо через `sizeof()`. Всередині циклу використовуємо лічильник як індекс масиву, щоб звернутися до кожного його осередку як `[i]`.

Такий цикл для роботи з масивом можна записати інакше:

```
for (тип_даних_масиву змінна: масив) {}
```

```
int vals[] = {10, 11, 12, 13, 14};
for (int val : vals) {
    Serial.println(val);
}
```

ми створюємо змінну `val` такого ж типу як масив, а також вказуємо ім'я масиву через двокрапку. На кожній ітерації циклу змінна `val` прийматиме значення осередку масиву в порядку від 0 до розміру масиву з кроком 1.

Таким чином ми вирішили те саме завдання, але написали менше коду. Важливий момент: на кожній ітерації циклу значення осередку надається до змінної `val`, тобто фактично ми можемо лише прочитати значення (через буферну змінну). Для безпосереднього доступу до елементів масиву потрібно створювати посилання, тобто просто додати оператор `&`

Цикли. Цикл for each

```
int vals[] = {10, 11, 12, 13, 14};  
for (int val : vals) {  
    Serial.println(val);  
}
```

val у цьому разі надає повний доступ до елементу масиву, тобто його читати/писати.

Приклад вище виведе значення кожного елемента, а потім обнуляє його. Після виконання циклу весь масив буде забитий нулями. Відсутність індексації в циклі може бути незручною для деяких алгоритмів, але лічильник завжди можна додати свій. Наприклад заб'ємо масив числами від 0 до 90 з кроком 10:

```
int vals[10];  
int count = 0;  
for (int &val : vals) {  
    val = count * 10;  
    count++;  
}
```

І це буде все ще компактніше класичного **for**.

Цикли. Оператор break

Оператор **break** дозволяє достроково покинути цикл, використовувати його можна як за умовою, так і як-зручно.

Приклад (достроково виходимо з циклу при досягненні якогось значення)

```
for (int i = 0; i < 100; i++) {  
    // залишити цикл при досягненні 50  
    if (i == 50) break;  
}
```

Приклад (залишаємо «вічний» цикл при натисканні на кнопку)

```
for(;;) {  
    if (кнопка натиснута) break;  
}
```

Оператор **break** є не єдиним варіантом виходу із циклу, є оператор пропуску – **continue**

Цикли. Оператор `continue`

Оператор `continue` достроково завершує поточну ітерацію циклу і переходить до наступної.

Приклад (заповнимо масив, як робили раніше, але пропускаємо один елемент)

```
// Створимо масив на 100 осередків
byte myArray[100];

for (int i = 0; i < 100; i++) {
    // якщо i дорівнює 10, пропускаємо
    if (i == 10) continue;

    // заповнимо всі комірки значенням 25
    myArray[i] = 25;
}
```

Таким чином, елемент під номером 10 не отримає значення 25, ітерація завершиться до операції присвоєння.

Цикли. Цикл while

Цикл **while**, він називається цикл із передумовою, виконується до того часу, поки вірна зазначена умова. Якщо умова одразу неправильна, цикл навіть не почне свою роботу і буде повністю пропущений. Оголошується дуже

просто:

```
int i = 0;
while (i < 10) {
    i++;
}
```

Це повний аналог циклу **for** з налаштуваннями (*int i = 0; i < 10; i ++*). Єдина відмінність у тому, що на останній ітерації **i** набуде значення 10, тому що на значенні 9 цикл дозволить виконання.

Ще цікавий варіант, який можна зустріти на просторах чужого коду. Працює на основі того факту, що будь-яке число, крім нуля, обробляється логікою як **true**:

```
byte a = 5;
while (a--) {
    // Виконається 5 разів
}
```

Цикл **while** теж зручно використовувати як вічний цикл, наприклад, очікуючи настання будь-якої події (натискання кнопки):

```
// виконується, доки не натиснута кнопка
while (кнопка не натиснута);
```

Цикли. Цикл while

Поки умова не станеться, код не піде далі, застрягне на цьому циклі. Як вже зрозуміло, оператор **if** тут не потрібен, потрібно вказувати саме логічне значення, можна навіть так:

```
while (true);
```

Все буде виконуватися тут нескінченно! Якщо не передбачимо **break**.

Крім циклу з передумовою, є ще цикл з постумовою, так званий **do while**. **do while** – «виконувати поки...», робота цього циклу повністю аналогічна циклу **while** за тим винятком, що умова задається після циклу, тобто, цикл виконається один раз, потім перевірить умову, а чи не змінилося щось. Приклад:

```
do {  
    // Тіло циклу  
} while (умова);
```


Лекцію закінчено
Дякую за увагу

