C VAGY C++?

TUDATOS SZOFTVERTERVEZÉS MIKROVEZÉRLŐKRE

Felhasznált és ajánlott irodalom

-
- Umann Kristóf: Programozási Nyelvek: C++
- Richard L. Halterman: Fundamentals of Programming C++
- Federico Busato: Modern C++ Programming – (C++11/14/17/20/23)
- Changkun Ou: Modern C++ Tutorial – (C++11/14/17/20)
- Shabaz: Arduino Library Creation and Testing Workflow

C vagy C++ a beágyazott környezetben

- Az Arduino-környezetben írt programok első pillantásra C nyelvűnek tűnhetnek, de a háttérben valójában mindig C++ fordító fut, mégis óriási a különbség aközött, hogy a nyelvet csak „szebb C”-ként használjuk, vagy tudatosan kiaknázzuk az objektumorientált (OOP) és típusbiztonsági lehetőségeit

C

```
// Globális állapotok
const int ledPin = 13;
bool ledState = false;

// Különálló függvény
void toggleLED() {
  ledState = !ledState;
  digitalWrite(ledPin, ledState);
}

void setup() {
  pinMode(ledPin, OUTPUT);
}
```

C++

pin = p inicializálás

```
class Indicator {
  private:
    const int pin;
    bool state = false;
  public:
    Indicator(int p) : pin(p) {}
    void init() { pinMode(pin, OUTPUT); }
    void toggle() { state = !state;
         digitalWrite(pin, state);
    }
};
Indicator led(13);
void setup() { led.init(); }
```

Mit nyerünk a C++ megközelítéssel?

- Bár a jobb oldali (C++) kód első pillantásra hosszabbnak tűnik a deklaráció miatt, néhány kulcsfontosságú dologra érdemes felfigyelni:
  - **Nincsenek globális változók:** A C++ változatban a **state** (állapot) és a **pin** változók rejtve vannak az objektumon belül (private). Senki nem tudja kívülről véletlenül felülírni vagy elrontani ezek tartalmát
  - **Skálázhatóság:** Ha a C kódban be kell vezetni egy második LED-et, le kell másolni a változókat (ledState2), és a függvényt is át kell írni vagy paraméterezni kell. C++-ban a bővítés mindössze egyetlen plusz sor a **setup()** előtt: `Indicator led2(12);.`
  - **Nulla rezsiköltség:** Megnyugtató, hogy a fordító ebből a strukturált C++ kódból ugyanakkora és ugyanolyan gyors gépi kódot generál, mint a bal oldali C változatból
- Ezek a tulajdonságok indokolják, hogy miért érdemes a C++ struktúrát választani, ha összetettebbé válik a projekt

Erőforrás-korlátok és C++ mozgástér

- **ATmega328P (Arduino UNO / Nano)**
  - Erőforrás: 2 KB RAM / 32 KB Flash *(Szűkös mikrovilág)*
  - C++ Játékszabály: Csak statikus objektumok. A dinamikus memóriakezelés (new, String) ellenjavalt. A C++ itt elsősorban a kód tisztább szervezését szolgálja
- **RP2040 (Raspberry Pi Pico)**
  - Erőforrás: 264 KB RAM / 2-16 MB Flash *(Kényelmes középút)*
  - C++ Játékszabály: Kiváló terep az absztrakcióra. Hardveres perifériák tiszta lekezelése dedikált osztálypéldányokkal. A RAM-mal nem kell filléreskedni, de a heap-fragmentációra ügyelni kell
- **ESP32 és ESP32-C3**
  - Erőforrás: 400-520 KB RAM / 4 MB Flash *(Beágyazott bőség)*
  - C++ Játékszabály: Itt a C++ már elengedhetetlen (a gyári WiFi és BLE stackek eleve objektumorientáltak). A modern C++ eszközök (például a lambda függvények) kényelmessé teszik az aszinkron eseménykezelést (callback-ek). Ugyanakkor a háttérben futó FreeRTOS többszálú környezetet teremt, így az objektumok tervezésekor ügyelni kell a szálbiztosságra (thread-safety)
- **Konklúzió**: nem lehet egy kaptafára venni a mikrovezérlőket, amikor C++-ról beszélünk

1. példa: Többcsatornás szenzorkezelés

- Van három analóg hőmérőnk: szobahőmérő, vízhőmérő a bojlernél, kültéri hőmérő. Be kell olvasni őket, mindegyikhez külön zajszűrést tervezünk, ráadásul a kazánra tett szenzornak gyorsan kell reagálnia a változásra, míg a kültérinek lassan, kiszűrve a szél és a zaj okozta ingadozásokat. Hasonlítsuk össze a C és C++ nyelvű megvalósítást!
- **A mérnöki feladat:**
  - Három MCP9700A analóg hőmérő szenzor egyidejű kezelése az A0, A1, A2 analóg bemeneteken
  - A mért adatok feszültséggé és Celsius-fokká alakítása (500 mV offset, 10 mV/℃ meredekség)
  - Csatornánkénti egyedi szoftveres zajszűrés (exponenciális mozgóátlag)
- **Mit akarunk ezzel bemutatni?**
  - **A C-megközelítés korlátait:** Hogyan zilálják szét a kód szerkezetét az elszigetelt globális tömbök, a nehezen követhető indexelések és a kódmásolás
  - **A C++ valódi erejét:** Hogyan foglalható egységbe a hardveres láb, a matematikai számítás és a szűrő egyedi paramétere és memóriája
  - **A konfigurálhatóságot:** Hogyan kaphat három teljesen azonos típusú szenzor különböző működési paramétereket (gyors vs. lassú szűrés) pusztán a létrehozás pillanatában

1. példa: Három hőmérő – a C program

```
// C megközelítés: minden adat külön tömbökben van
const int sensorPins[3] = {A0, A1, A2};
float lastValues[3] = {0.0, 0.0, 0.0};
const float alpha = 0.2;          // Szűrő együttható

float readMCP9700(int index) {
  int raw = analogRead(sensorPins[index]);
  float voltage = (raw * 5.0) / 1023.0;
  float currentTemp = (voltage - 0.5) * 100.0;

  // Szűrés
  lastValues[index] = (alpha * currentTemp) + ((1.0 - alpha) * lastValues[index]);
  return lastValues[index];
}

void setup() {
  Serial.begin(9600); // A kommunikáció kapuját ki kell nyitni!
}

void loop() {
  for(int i = 0; i < 3; i++) {
    Serial.print(readMCP9700(i));  // Ki vagyunk szolgáltatva az indexeknek
  } // A hardver és a szoftver között nincs szoros kapcsolat, csak egy törékeny számozás
}
```

Három hőmérő – a C++ program

```
class MCP9700 {
  private:
    int pin; float filteredTemp = 25.0; // Kezdőérték
    float alpha;                // Egyedi szűrési tényező
  public:
    MCP9700(int p, float a = 0.1) : pin(p), alpha(a) {}     // Konstruktor: beállítja a lábat és a szűrőt
    void update() {
      int raw = analogRead(pin);  float voltage = (raw * 5.0) / 1023.0;
      float rawTemp = (voltage - 0.5) * 100.0;
      filteredTemp = (alpha * rawTemp) + ((1.0 - alpha) * filteredTemp);      // Exponenciális simítás
    }
    float getTemperature() const { return filteredTemp; }
};
MCP9700 roomSensor(A0, 0.1);          // Statikus példányosítás
MCP9700 boilerSensor(A1, 0.3);        // Ennek gyorsabban kell reagálnia (más az alpha!)
MCP9700 outdoorSensor(A2, 0.05);      // Ez lassan változik, erős szűrés kell
void setup() { Serial.begin(9600); }
void loop() {
  roomSensor.update(); boilerSensor.update(); outdoorSensor.update();
  Serial.print("Szoba: ");  Serial.println(roomSensor.getTemperature());
  Serial.print("Kazán: ");  Serial.println(boilerSensor.getTemperature());
  Serial.print("Kültér: "); Serial.print(outdoorSensor.getTemperature()); Serial.println(" °C");
  Delay(1000); }
```

2. példa: Periféria-absztrakció és az inline titka

- **A mérnöki feladat**
  - Egy RGB LED fényerejének (vagy egy motor sebességének) szabályozása hardveres PWM segítségével.
  - **A cél:** Kiváltani a kényelmes, de a háttérben nehézkes, futásidejű ellenőrzéseket és keresőtáblákat használó Arduino analogWrite() függvényt egy sokkal hatékonyabb megközelítéssel.
- **A C++ attrakció:** Az egységbezárás (Encapsulation) és az inline tagfüggvények kombinációja
  - A hardver konfigurálását áthelyezzük a program indulásakor csak egyszer lefutó konstruktorba
  - A fényerő állításakor az *inline* kulcsszó miatt az osztályba zárt kód a fordítás után eltűnik, és a fordító a háttérben egyetlen közvetlen regiszterírássá alakítja a hívást
  - Az absztrakciónak így nulla időbeli overheadje (Zero-cost abstraction) lesz, vagyis úgy kapunk tiszta, olvasható kódot, hogy a processzor szintjén maximális sebességet érünk el

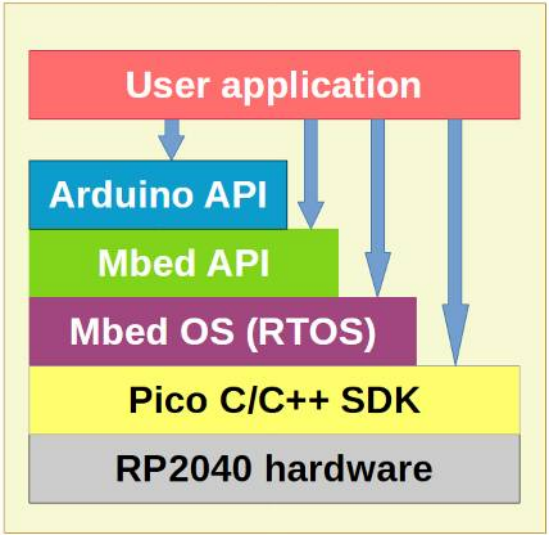
PWM: RP2040 megközelítés (Pico SDK hívásokkal)

```
#include "hardware/pwm.h"
#include "hardware/gpio.h"

class SmartPWM {
  private:
    uint pin, slice_num, chan; // Változók
  public:
    SmartPWM(uint gpio_pin) : pin(gpio_pin) { // Konstruktor
        gpio_set_function(pin, GPIO_FUNC_PWM);
        slice_num = pwm_gpio_to_slice_num(pin);
        chan = pwm_gpio_to_channel(pin);
        pwm_set_wrap(slice_num, 255);          // 8 bites felbontás (~1 kHz)
        pwm_set_enabled(slice_num, true);
    }
    inline void setBrightness(uint8_t value) { pwm_set_chan_level(slice_num, chan, value); }
};

SmartPWM ledRed(16),ledGreen(17),ledBlue(18); // Példányosítás

int main() {
    ledRed.setBrightness(128);                 // 50% fényerő
    ledGreen.setBrightness(0);                 // Kikapcsolva
    ledBlue.setBrightness(255);                // 100% fényerő
    while(1) { /* Háttérfolyamatok... */ }
}
```



PWM: ATmega328 megközelítés – 2/1

```
#include <avr/io.h>

class AtmegaPWM16 {
  private:
    bool isChannelB; // Eldönti futási időben, hogy a 9-es vagy 10-es lábról van szó
  public:
    AtmegaPWM16(uint8_t pin) {      // Konstruktor: konfigurálja a hardveres Timer 1-et
        if (pin == 9) {
            isChannelB = false; DDRB |= (1 << PB1); // 9-es láb (OC1A) kimenetre állítása
        } else if (pin == 10) {
            isChannelB = true;  DDRB |= (1 << PB2); // 10-es láb (OC1B) kimenetre állítása
        }
        TCCR1A = (1 << WGM10);      // Timer 1 konfigurálása:
        TCCR1B = (1 << WGM12) | (1 << CS11) | (1 << CS10); // Előosztó: 64 (~980 Hz)
    }
    inline void setBrightness(uint8_t value) {
        if (!isChannelB) { TCCR1A |= (1 << COM1A1); OCR1A = value; }
        Else { TCCR1A |= (1 << COM1B1); OCR1B = value;  }
    }
};
```

PWM: ATmega328 megközelítés – 2/2

```
// Példányosítás a két fix Timer 1-es lábra
AtmegaPWM16 ledRed(9);           // OC1A
AtmegaPWM16 ledGreen(10);        // OC1B

int main() {
    // A főprogram pontosan ugyanolyan szép, mint az RP2040-nél!
    ledRed.setBrightness(128);   // 50% fényerő
    ledGreen.setBrightness(255); // 100% fényerő

    while(1) {
        // Háttérfolyamatok...
    }
}
```

```
Atmega: Sketch uses 290 bytes (0%) of program storage space. Maximum is 30720 bytes.
Global variables use 2 bytes (0%) of dynamic memory, leaving 2046 bytes for local variables. Maximum is
2048 bytes.

RP2040/mbed: Sketch uses 89842 bytes (4%) of program storage space. Maximum is 2097152 bytes.
Global variables use 42872 bytes (15%) of dynamic memory, leaving 227464 bytes for local variables.
Maximum is 270336 bytes - a nagy méret a befordított libmbed.a miatt van.
```

Konklúzió

- **Mit NYERÜNK a C++-szal?**
  - Brutális sebességet (Hatékonyságot): Mivel elhagytuk a futásidejű lookup-táblázatokat, az elágazásokat és a függvényhívás overheadjét, a kódunk 30-40-szer gyorsabban hajtja végre a fényerőváltoztatást, mint az **analogWrite()** függvény
  - Kiszámíthatóságot: Nincsenek rejtett háttér-függvények. Pontosan tudjuk, melyik regiszter mikor változik
- **Mit VESZTÜNK a C++-szal?**
  - **Rugalmasságot:** Az **analogWrite()**-nál a láb számát egy változóból is megadhatjuk, ami a program futása közben változik (pl. `analogWrite(valtozo_lab, 128))` A mi C++ osztályunknál a láb száma fixen be van drótozva a példányosításkor. Futási időben már nem mondhatjuk azt a **ledRed**-nek, hogy mostantól a 10-es lábat vezérelje
  - **Hordozhatóságot**: Az **analogWrite()** ugyanúgy lefut egy ATmega328p-n és egy RP2040-en. A mi osztályunk viszont – mivel a maximális sebesség érdekében közvetlenül a chip saját hardverregisztereit (OCR1A vagy Pico SDK függvények) használja – nem hordozható. Ha ATmega-ról RP2040-re váltunk, az osztály belső regiszteríró kódját teljesen újra kell írni

3. példa: A WS2812 (Neopixel) meghajtása

- A címezhető LED-ek (WS2812) villogtatása szoftveres szempontból egy rémálom. A bitek kiküldésének időzítése olyan szigorú toleranciával dolgozik, hogy ha a processzor csak egy pillanatra is félrenéz (például lefut egy megszakítás), a LED-csík azonnal megbolondul
- **A mérnöki feladat:**
- Megmutatjuk, hogyan lehet a modern C++ eszközökkel (fordítási idejű absztrakcióval) „kettéválasztani a világot”:
  - **A hardverfüggetlen felszín:** Írunk egy tiszta alkalmazást (egy szivárvány-effektet), amelynek fogalma sincs arról, milyen mikrovezérlőn fut
  - **A motorháztető alatti valóság:** Készítünk egy intelligens könyvtárat, ami az **ATmega328P**-n ciklusra pontos inline assembly-vel rángatja a lábat, míg az **RP2040**-en átadja a munkát a processzortól független PIO koprocesszornak

WS2812_demo.ino

```
#include "SmartWS2812.h"
RGB strip[8];
uint8_t startColor = 0;

#if defined(__AVR_ATmega328P__)
  SmartWS2812 leds(8);              // Példányosítás: Atmegánál D8
#else
  SmartWS2812 leds(16);             // RP2040-Zero: a beépített LED
#endif

int main() {
    while(1) {
        for(uint8_t i = 0; i < 8; i++) {
            strip[i] = wheel(startColor + (i * 32));
        }
        leds.show(strip, 8);
        startColor -= 32;
        DRIVER_DELAY_MS(1000);    // 1 Hz-es léptetés
    }
}
```

SmartWS2812_demo.h – 3/1.

```
#ifndef SMART_WS2812_H
#define SMART_WS2812_H
#include <stdint.h>
struct RGB { uint8_t r, g, b; };       // Közös adattípus a színeknek

inline RGB wheel(uint8_t pos) {
    if (pos < 85)  return RGB{uint8_t(pos * 3), uint8_t(255 - pos * 3), 0};
    if (pos < 170) { pos -= 85; return RGB{uint8_t(255 - pos * 3), 0, uint8_t(pos * 3)}; }
    pos -= 170;    return RGB{0, uint8_t(pos * 3), uint8_t(255 - pos * 3)};
}
#if defined(__AVR_ATmega328P__)        // 1. ÁG: ATmega328P (Arduino Uno / Nano) implementáció
#include <avr/io.h>
#include <util/delay.h>
#define DRIVER_DELAY_MS(x) _delay_ms(x)

class SmartWS2812 {
  public:
    SmartWS2812(uint8_t pin) {
        DDRB |= (1 << PB0);            // Oktatói egyszerűsítés: fixen a D8-as (PB0) lábat használjuk
    }

    void show(RGB* leds, uint8_t count) {
        cli();                         // Megszakítások tiltása a kritikus időzítés miatt
        for (uint8_t i = 0; i < count; i++) {
            sendByte(leds[i].g); sendByte(leds[i].r); sendByte(leds[i].b);
        }
        Sei(); _delay_us(50);          // Reset jel
    }
```

SmartWS2812_demo.h – 3/2.

```
  private:
    void sendByte(uint8_t byte) {
        for (uint8_t bit = 0; bit < 8; bit++) {
            if (byte & 0x80) {
                PORTB |= (1 << PB0);  asm volatile("nop; nop; nop; nop; nop;");
                PORTB &= ~(1 << PB0); asm volatile("nop;");
            } else {
                PORTB |= (1 << PB0);  asm volatile("nop;");
                PORTB &= ~(1 << PB0); asm volatile("nop; nop; nop; nop; nop;");
            }
            byte <<= 1;
        }
    }
};

// ============================================================================
// 2. ÁG: RP2040 (Raspberry Pi Pico) implementáció
// ============================================================================
#elif defined(ARDUINO_ARCH_RP2040)
#include "hardware/pio.h"
#include "ws2812.pio.h"
#define DRIVER_DELAY_MS(x) sleep_ms(x)
```

SmartWS2812_demo.h – 3/3.

```
class SmartWS2812 {
  private:
    PIO pio = pio0; uint sm = 0;
  public:
    SmartWS2812(uint pin) {
        uint offset = pio_add_program(pio, &ws2812_program);
        ws2812_program_init(pio, sm, offset, pin, 800000, false);
    }

    void show(RGB* leds, uint8_t count) {
        for (int i = 0; i < count; i++) {
            uint32_t data = ((uint32_t)leds[i].g << 16) | ((uint32_t)leds[i].r << 8) | leds[i].b;
            pio_sm_put_blocking(pio, sm, data << 8);
        }
    }
};

// ============================================================================
// VÉDELEM: Nem támogatott hardver esetén kapásból elhasal a fordítás
// ============================================================================
#else
  #error "Nem támogatott hardver! Ez a könyvtár csak ATmega328P és RP2040 chipeken működik!"
#endif

#endif
```

Mit tanultunk a 3. mintapéldából?

- A feltételes fordítással és az *inline* függvényekkel sikerült egy olyan szoftverarchitektúrát létrehoznunk, ami úgy hordozható, hogy közben nem fizettünk érte semmilyen árat. Az ATmega328P ágán nem keletkezett virtuális függvénytábla (vtable), a kód pontosan olyan pici, gyors és optimalizált maradt, mintha eleve csak az Uno-ra írtuk volna meg a nyers assembly-t
- Bár a 8 LED-es szivárvány mindkét kártyán ugyanúgy néz ki, a motorháztető alatt drámai a különbség a CPU kihasználtságában:
  - **ATmega328P:** A LED-ek frissítése közben a processzor 100%-ban blokkolt, a megszakításokat le kellett tiltani. Ha a csík nem 8, hanem 800 LED-ből állna, az Uno az idő nagy részében teljesen „vak” lenne a külvilágra.
  - **RP2040:** A CPU csak átdobja az adatokat a FIFO pufferbe, a fizikai jelsorozatot pedig a PIO állapotgép generálja a háttérben. A processzor magjai azonnal szabadok, így mialatt a PIO a LED-eket villogtatja, a CPU-val párhuzamosan olvashatnánk szenzorokat vagy adatforgalmat kezelhetnénk

4. példa: Programkönyvtár az ST7585 kijelzőhöz

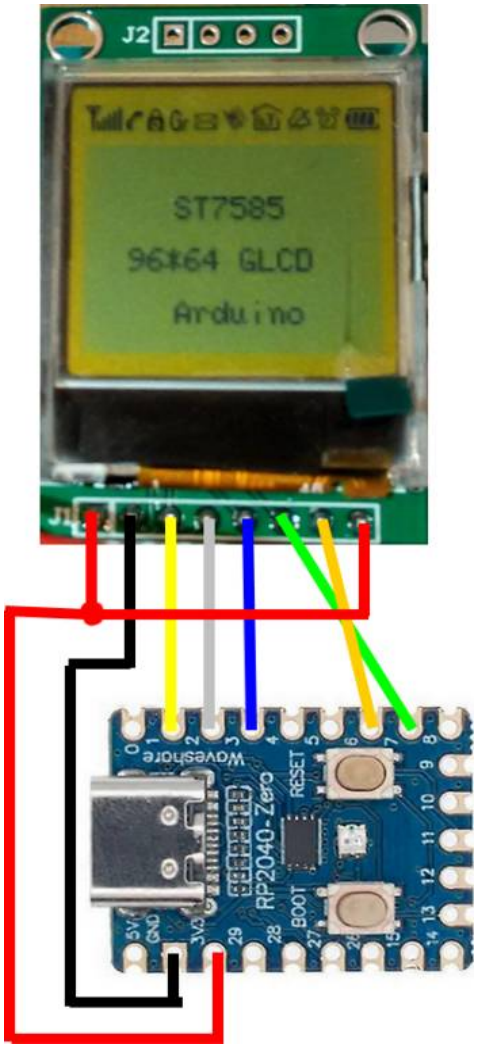
- Az **ST7585_GFX** könyvtár az **Adafruit_GFX** osztályból örökölve teremti meg a kapcsolatot az absztrakt grafikus függvények és a monokróm, 96×64 képpontos LCD között
- A fejlesztés során implementálni kell az SPI busz és a GPIO vezérlővonalak (CS, RST, DC) platformfüggetlen konfigurációját, valamint a kijelző saját parancskészletére épülő inicializálást
- A grafikus tartalom kezeléséhez csak egy memóriabuffert (Page 0–7) és a **drawPixel()** függvényt kell biztosítanunk
- Ennek a kijelzőnek a tetején található egy fix ikonsor is, ennek elérését a **writeIconByte()** függvény biztosítja (Page 8)

Shanyan kijelző 96x64 pixel ST7585 vezérlő

- A C++ objektumorientált architektúra fő előnye itt az adatrejtésben és a kód-újrafelhasználásban rejlik. A C++ öröklődés (inheritance) mechanizmusának köszönhetően az ősosztály összes magas szintű rajzoló- és szövegkezelő függvénye (mint a **print(), drawLine(), drawRoundRect()**) automatikusan működnek/használhatók

Kivezetések és a bekötési vázlat

- VDD (Voltage Drain Drain) tápfeszültség (+3,3 V)
- GND (Ground) a tápegység közös pontja
- CS (Chip Select) modul aktiválásához alacsony szintre kell húzni
- RST (Reset) a modult alaphelyzetbe állítja
- D/C (Data/Command) adat vagy parancsküldés választó vonal
- SDI (Serial Data Input) SPI soros adatbemenet
- SCK (Serial Clock) SPI órajel
- BL (Backlight) LED megvilágítás (If = 20 mA, Uf = 2,1 V typ.



ST6585_GFX.h

```
#ifndef _ST7585_GFX_H
#define _ST7585_GFX_H

#include "Arduino.h"
#include <Adafruit_GFX.h>
#include <SPI.h>

#define ST7585_BLACK 1
#define ST7585_WHITE 0

#define ST7585_LCDWIDTH  96
#define ST7585_LCDHEIGHT 64

#define ST7585_FUNCTIONSET       0x20
#define ST7585_DISPLAYCONTROL    0x08
#define ST7585_SETYADDR          0x40
#define ST7585_SETXADDR          0x80
#define ST7585_EXTENDEDINSTRUCTION 0x01
#define ST7585_SETTESTMODE         0x03
#define ST7585_SETVOP              0x80
#define ST7585_DISPLAYNORMAL     0x04
```

```
class ST7585_GFX : public Adafruit_GFX {
 public:
  ST7585_GFX(SPIClass *spiClass, int8_t dc_pin,
         int8_t rst_pin, int8_t cs_pin = -1);

  void begin(uint8_t contrast = 0x30);
  void setContrast(uint8_t val);
  void clearDisplay();
  void display();

  void drawPixel(int16_t x,int16_t y,uint16_t color) override;
  uint8_t getPixel(int16_t x, int16_t y);
  void writeIconByte(uint8_t column, uint8_t data);
  void command(uint8_t c);
  void data(uint8_t c);

 private:
  SPIClass *_spi; // Mutató a felhasznált SPI objektumra
  int8_t _dc, _rst, _cs;
  uint8_t buffer[ST7585_LCDWIDTH * ST7585_LCDHEIGHT / 8];
  SPISettings spiSettings;
};

#endif
```

ST7585.cpp (részlet)

- Konstruktor és inicializálás

```
#include "ST7585_GFX.h"

ST7585_GFX::ST7585_GFX(SPIClass *spiClass, int8_t dc_pin, int8_t rst_pin, int8_t cs_pin)
  : Adafruit_GFX(ST7585_LCDWIDTH, ST7585_LCDHEIGHT), spiSettings(4000000, MSBFIRST, SPI_MODE0) {
  _spi = spiClass;
  _dc = dc_pin;
  _rst = rst_pin;
  _cs = cs_pin;
}

void ST7585_GFX::begin(uint8_t contrast) {
  pinMode(_dc, OUTPUT);
  if (_rst >= 0) pinMode(_rst, OUTPUT);
  if (_cs >= 0) pinMode(_cs, OUTPUT);
  if (_rst >= 0) {
    digitalWrite(_rst, LOW);     delay(10);
    digitalWrite(_rst, HIGH);    delay(10);
  }
  command(ST7585_FUNCTIONSET | ST7585_EXTENDEDINSTRUCTION);
  setContrast(contrast);
  command(ST7585_SETTESTMODE | 0x02);
  command(ST7585_FUNCTIONSET);
  command(ST7585_DISPLAYCONTROL | ST7585_DISPLAYNORMAL);
  clearDisplay();
}
```

ST7585.cpp (részlet)

- **drawPixel()** és **getPixel()** megvalósítása

```
void ST7585_GFX::drawPixel(int16_t x, int16_t y, uint16_t color) {
  if ((x < 0) || (x >= _width) || (y < 0) || (y >= _height)) return;

  int16_t t;
  switch (rotation) {
    case 1: t = x; x = y; y = ST7585_LCDHEIGHT - 1 - t; break;
    case 2: x = ST7585_LCDWIDTH - 1 - x; y = ST7585_LCDHEIGHT - 1 - y; break;
    case 3: t = x; x = ST7585_LCDWIDTH - 1 - y; y = t; break;
  }

  if ((x < 0) || (x >= ST7585_LCDWIDTH) || (y < 0) || (y >= ST7585_LCDHEIGHT)) return;

  if (color) buffer[x + (y / 8) * ST7585_LCDWIDTH] |= (1 << (y % 8));
  else       buffer[x + (y / 8) * ST7585_LCDWIDTH] &= ~(1 << (y % 8));
}

uint8_t ST7585_GFX::getPixel(int16_t x, int16_t y) {
  if ((x < 0) || (x >= ST7585_LCDWIDTH) || (y < 0) || (y >= ST7585_LCDHEIGHT)) return 0;
  return (buffer[x + (y / 8) * ST7585_LCDWIDTH] >> (y % 8)) & 0x01;
}
```

Az ikonsor kezelése

- Az ikonsor a kijelző extra sávján (Page 8) lakik. A **writeIconByte()** az adatbájtot pufferekeés nélkül, közvetlenül küldi el a kijelző vezérlőjének, az SPI buszon.
  A függvény egy oszlopindexet (X=0…95) és egy 8 bites maszkot vár. Egy-egy ikon bekapcsolásához több egymás melletti oszlopba kell a megfelelő bitmintát beírni.

```
void ST7585_GFX::writeIconByte(uint8_t column, uint8_t data_byte) {
   if (column >= ST7585_LCDWIDTH) return;
   // 1. Kapcsoljunk be biztosan a normál parancsmódba
   command(0x20);
   // 2. Küldjük el a Page 8 parancsot fixen (az adatlap szerint az ikonsor címe: 0x48)
   command(0x48);
   // 3. Küldjük el az oszlop címét (0x80 | oszlop)
   command(0x80 | column);
   // 4. Mehet az adat!
   data(data_byte);
   // 5. Visszaléptetés a normál zónába (Page 0), hogy a display() ne akadjon ki
   command(0x40);
}
```

ST7585_demo.ino

- A kód a C++ preprocesszor makrói segítségével fordítási időben automatikusan felismeri a célarchitektúrát (AVR Arduino, ESP32, ESP32-C3, vagy RP2040) és ennek megfelelően konfigurálja az adott platform SPI lábait és indítja el a buszt
- **Kétlépcsős hardverteszt** (Setup fázis):
  - **Grafikus és geometriai teszt:** Első lépésben egy rácsmintával, teljes ikonsor-kivilágítással, majd az Adafruit_GFX motorjára épülő dinamikus vonalrajzzal ellenőrzi a Page 0–7 (grafikus) és a Page 8 (ikon) területek fizikai épségét
  - **Intelligens üdvözlőképernyő:** A tesztek lefutása után a kijelzőn megjelenik egy szoftveres keret, és nagy méretű betűkkel kiíródik a szoftver által detektált mikrovezérlő neve (pl. RP2040 vagy ESP32-C3).
- **Futófény effekt** (Loop fázis): A fő hurokban egy végtelenített futófény pörög az ikonsoron. A **writeIconByte()** függvény segítségével, közvetlen SPI parancsokkal, oszlopról oszlopra kapcsolja be, majd törli ki az ikonok bitjeit