C++ v embedded systémech

Arduino · ESP32 · STM32

Stanislav Vítek · Katedra radioelektroniky · ČVUT FEL

Proč C++ na mikrokontroléru?

Tradičně se firmware psal v C — blízko hardwaru, předvídatelné chování, žádný skrytý overhead. C++ mělo pověst jazyka, který „tajně alokuje paměť" a „přidává skrytá volání". Tato pověst byla kdysi částečně oprávněná. Dnes ne.

Moderní C++ (C++11 a novější) přináší nástroje, které jsou **nulově nákladné za běhu** — vše se odehraje při překladu:

- **Šablony** — generický, typově bezpečný kód bez runtime overhead
- `constexpr` — výpočty přesunuté do compile time → konstanty v ROM místo RAM
- **RAII** — automatická správa zdrojů bez garbage collectoru
- `inline` a agresivní optimalizace překladačem

> ***Zero-cost abstraction:*** *pokud to nezaplatíš, nezaplatíš za to nic.*
> *Pokud to zaplatíš, nemohl bys to napsat lépe ručně.*
> *— Bjarne Stroustrup*

Výsledek: firmware je **čitelnější a bezpečnější** než C, ale stejně rychlý a paměťově stejně náročný.

C++ vs. C vs. MicroPython

C

- Maximum kontroly
- Přímý přístup k HW
- Žádná abstrakce — vše ručně
- Nebezpečné: buffer overflows, null pointery
- Vhodné: RTOS kernely, ovladače na úrovni registrů

MicroPython / CircuitPython

- Extrémně rychlý vývoj
- Interpret → 10–100× pomalejší
- Heap, GC → nepředvídatelné latence
- RAM overhead ~50–100 kB jen pro interpret
- Vhodné: prototypy, výuka základů

C++ (moderní, embedded styl)

- Abstrakce bez runtime ceny
- Typová bezpečnost, RAII
- Šablony → generický HW kód
- Přímý přístup k HW stále možný
- Vhodné: produkční firmware, knihovny

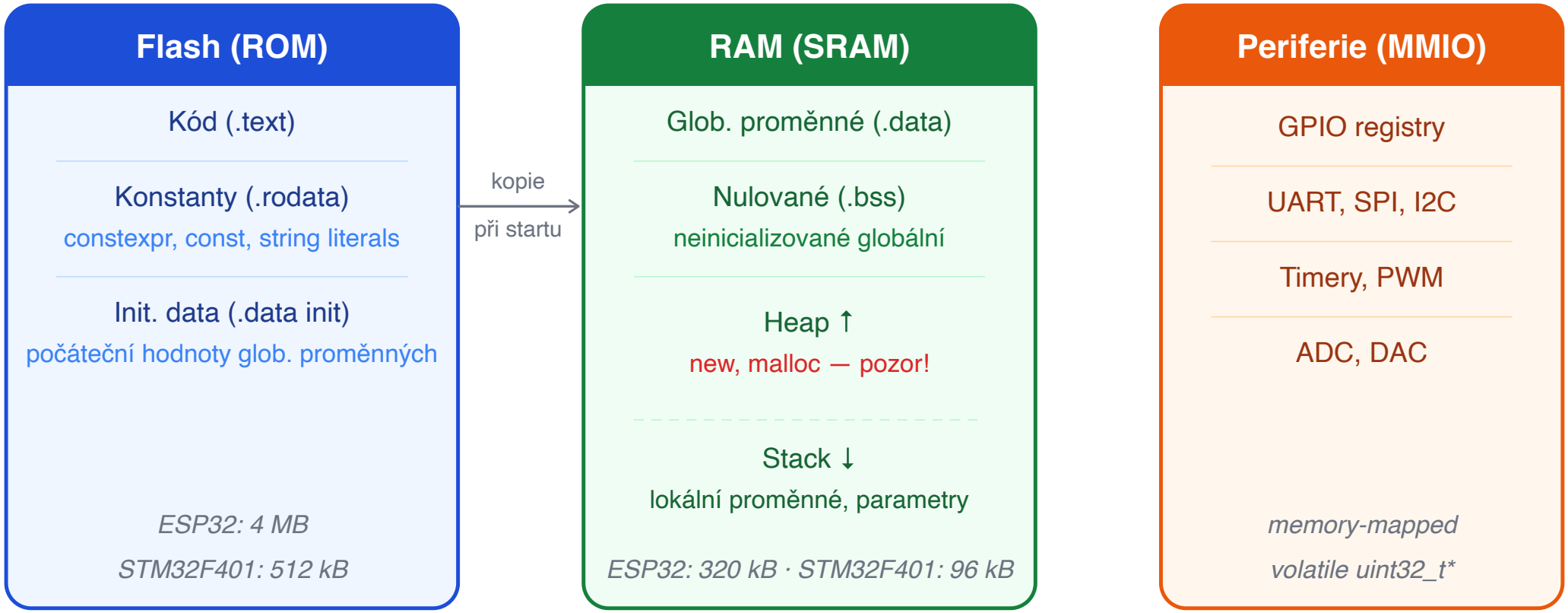
Konkrétní volba záleží na:

- Paměti (RAM/Flash): <8 kB → spíše C
- Týmu: Python background → MicroPython pro prototyp, pak C++
- Požadavcích na latenci: real-time → C nebo C++

Arduino API je napsáno v C++ — `pinMode()`, `digitalWrite()`, `Serial` jsou třídy a funkce v C++ namespace. Když píšete Arduino sketch, **already píšete C++.**

Paměťový model MCU

Pochopení paměťového modelu je klíčové — jinak C++ featura, která je na PC nevinná, může na MCU způsobit katastrofu.



Paměťový model — co to znamená v praxi

Tabulka shrnuje, kde skončí každý typ dat. Než napíšete kód na MCU, je dobré vědět, kam každá proměnná jde.

Konstrukce C++	Sekce	Paměť
`const char msg[] = "Hello"`	`.rodata`	Flash
`constexpr int BAUD = 115200`	`.rodata` nebo inline	Flash
`int globalCounter = 0`	`.bss`	RAM
`int globalInit = 42`	`.data`	RAM (+ Flash pro init hodnotu)
`static int x` (lokální)	`.bss` / `.data`	RAM
lokální proměnná ve funkci	zásobník	RAM (stack)
`new Foo()`	heap	RAM
`GPIOA->ODR`	MMIO	žádná — přímo periferie

Klíčové omezení: zásobník (stack) na MCU je typicky jen 4–8 kB. Hluboká rekurze nebo velká lokální pole (např. `uint8_t buf[4096]`) způsobí přetečení zásobníku — a na embedded to znamená tichý crash bez výjimky.

> ⚠ Na MCU neexistuje virtuální paměť. Přetečení zásobníku nebo heapu = přepsání jiných dat. Žádná ochrana, žádná výjimka — jen záhadné chování firmware.

`volatile` a přístup k registrům

Memory-Mapped I/O

Jak C++ komunikuje přímo s hardwarem

Memory-Mapped I/O — princip

Mikrokontroléry mapují registry periferií do adresového prostoru procesoru. GPIO pin nastavíte zápisem na konkrétní adresu — stejně jako do RAM. Periferie „vidí" tento zápis a reaguje.

```
// Adresa registru GPIO portu A na STM32F4 (z datasheets)
constexpr uint32_t GPIOA_BASE = 0x4002'0000;
constexpr uint32_t ODR_OFFSET = 0x14;          // Output Data Register

// Přímý přístup přes pointer na volatile uint32_t
volatile uint32_t* GPIOA_ODR =
    reinterpret_cast<volatile uint32_t*>(GPIOA_BASE + ODR_OFFSET);
```

Klíčové je slovo `volatile`. Bez něj by kompilátor mohl optimalizovat opakované přístupy — předpokládal by, že hodnota v paměti se nemění, pokud ji program sám nezměnil. U periferie to neplatí: hardware mění hodnotu nezávisle na procesoru.

```
// BEZ volatile — kompilátor může tyto řádky sloučit nebo vynechat:
*GPIOA_ODR = 0x0001;   // zapni pin
*GPIOA_ODR = 0x0000;   // vypni pin  ← kompilátor: "zbytečné, přepíšu hned"

// S volatile — KAŽDÝ zápis se skutečně provede:
volatile uint32_t* reg = reinterpret_cast<volatile uint32_t*>(0x40020014);
*reg = 0x0001;          // zapni
*reg = 0x0000;          // vypni — tentokrát se provede!
```

`volatile` — tři situace, kdy ho musíte použít

1. Registry periferií (MMIO)

Hardware mění hodnotu registru nezávisle na CPU — přečtení musí vždy sáhnout do skutečné paměti, ne do cache registru.

```
volatile uint32_t* UART_SR =
    reinterpret_cast<volatile uint32_t*>(0x40011000);

// Čekáme, než UART dokončí vysílání (bit TX_EMPTY = bit 7)
while (!(*UART_SR & (1u << 7))) { }   // volatile: čte pokaždé znovu
```

2. Sdílené proměnné s ISR

Přerušení může změnit proměnnou kdykoli — kompilátor to neví.

```
volatile bool tlacitkoStisknuto = false;   // měněno z ISR

void IRAM_ATTR buttonISR() { tlacitkoStisknuto = true; }

void loop() {
    if (tlacitkoStisknuto) {       // čte skutečnou RAM, ne registr CPU
        tlacitkoStisknuto = false;
        handlePress();
    }
}
```

Bitové operace na registrech (1/2)

Práce s jednotlivými bity registru je základní dovedností embedded programátora. Tři nejdůležitější operace:

```
volatile uint32_t* reg = reinterpret_cast<volatile uint32_t*>(0x40020014);

// Nastavit bit N (set) — ostatní bity zachovat
*reg |= (1u << N);

// Vymazat bit N (clear) — ostatní bity zachovat
*reg &= ~(1u << N);

// Přepnout bit N (toggle)
*reg ^= (1u << N);

// Přečíst bit N — výsledek je 0 nebo nenula
bool bitJeNastaveny = (*reg >> N) & 1u;
```

Bitové operace na registrech (2/2)

V praxi jsou piny a registry pojmenovány — nikdo nepíše `0x40020014` přímo do kódu. HAL knihovny (STM32 HAL, Arduino, ESP-IDF) poskytují makra a struktury:

```
// Arduino abstrakce – interně dělá totéž
pinMode(LED_BUILTIN, OUTPUT);
digitalWrite(LED_BUILTIN, HIGH);

// STM32 HAL – o úroveň níže, ale stále abstrakce
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);

// Přímý registrový přístup – maximální kontrola
GPIOA->ODR |= GPIO_PIN_5;    // GPIOA je volatile struct* z stm32f4xx.h
```

Bitová pole v C++ strukturách

Alternativou k bitovým maskám jsou bitová pole. Jsou čitelnější, ale mají záludnosti.

```
// Bitová pole – pěkně čitelné...
struct UartStatus {
    uint32_t rxFull   : 1;   // bit 0
    uint32_t txEmpty  : 1;   // bit 1
    uint32_t parError : 1;   // bit 2
    uint32_t          : 13;  // bity 3–15 nevyužity
    uint32_t overrun  : 1;   // bit 16
    uint32_t          : 15;  // zbytek
};

volatile UartStatus* status =
    reinterpret_cast<volatile UartStatus*>(UART_BASE);

// Použití – čitelné
if (status->rxFull) { readByte(); }
```

⚠ **Pořadí bitů v bitových polích není standardem garantováno** — závisí na překladači a architektuře. Na různých platformách může být jiné. Pro přenositelný kód je bezpečnější používat bitové masky. Bitová pole jsou v pořádku pro jednoplatformové projekty s dobře zdokumentovaným překladačem (GCC ARM).

RAII pro hardware

Resource Acquisition Is Initialization

Automatická správa HW zdrojů bez garbage collectoru

RAII — připomenutí principu

RAII (Resource Acquisition Is Initialization) je vzor, podle kterého se zdroj získá v konstruktoru a uvolní v destruktoru — automaticky, i při výjimce nebo předčasném návratu z funkce.

Na PC se RAII typicky používá pro soubory, mutexy a síťová spojení. Na MCU je stejně cenný pro **hardwarové zdroje**: GPIO piny, SPI transakce, kritické sekce, DMA kanály.

```
// Bez RAII — musíme myslet na uvolnění v každé větvi
void sendPacket() {
    SPI.beginTransaction(SPISettings(4e6, MSBFIRST, SPI_MODE0));
    digitalWrite(CS_PIN, LOW);

    if (error_condition) {
        digitalWrite(CS_PIN, HIGH);    // musíme explicitně uklidit
        SPI.endTransaction();
        return;
    }

    SPI.transfer(data, len);

    digitalWrite(CS_PIN, HIGH);        // a tady taky
    SPI.endTransaction();
}
```

Pokud přidáme další `return` nebo výjimku, snadno zapomeneme na `endTransaction()`. Výsledek: SPI bus zůstane zablokovaný a firmware přestane komunikovat.

RAII — GPIO pin (1/2)

Nejjednodušší příklad: třída `DigitalOutput`, která pin nastaví v konstruktoru a uvolní v destruktoru. Začneme speciálními metodami — ty určují, jak se třída chová při kopírování a přesouvání.

```
class DigitalOutput {
    uint8_t pin;
public:
    // Konstruktor – získání zdroje (příkaz periferii)
    explicit DigitalOutput(uint8_t p, bool initState = LOW)
        : pin(p) {
        pinMode(pin, OUTPUT);
        digitalWrite(pin, initState);
    }

    // Destruktor – uvolnění zdroje (pin vrácen do bezpečného stavu)
    ~DigitalOutput() { pinMode(pin, INPUT); }

    // GPIO pin se nesmí kopírovat – dva objekty by ovládaly jeden pin
    DigitalOutput(const DigitalOutput&)            = delete;
    DigitalOutput& operator=(const DigitalOutput&) = delete;

    // Přesun (move) je OK – vlastnictví pinu se předá novému objektu
    DigitalOutput(DigitalOutput&& other) noexcept
        : pin(other.pin) { other.pin = 255; }   // 255 = "žádný pin"
};
```

RAII — GPIO pin (2/2)

Přidáme metody pro ovládání pinu a ukážeme použití:

```
    // (pokračování třídy DigitalOutput)
    void set(bool val) { digitalWrite(pin, val); }
    void high()        { digitalWrite(pin, HIGH); }
    void low()         { digitalWrite(pin, LOW); }
    void toggle()      { digitalWrite(pin, !digitalRead(pin)); }
    bool read()  const { return digitalRead(pin); }
```

Použití — bez RAII bychom museli pamatovat na `pinMode(INPUT)` při každém ukončení:

```
void blikni(uint8_t pin, int n) {
    DigitalOutput led(pin);   // pin nastaven jako OUTPUT

    for (int i = 0; i < n; ++i) {
        led.high(); delay(200);
        led.low();  delay(200);
    }
}   // ← destruktor: pinMode(pin, INPUT) – automaticky

// Nelze omylem zkopírovat:
// DigitalOutput led2 = led;   // ✗ chyba překladu
```

Třída je nekopírovatelná — fyzický pin nemůžeme duplikovat. Přesouvatelná je — vlastnictví pinu lze předat, například do kontejneru.

RAII — SPI transakce jako guard (1/2)

Klasický RAII guard pro SPI: `beginTransaction` v konstruktoru, `endTransaction` v destruktoru. Uklizení je garantováno bez ohledu na to, jak funkce skončí.

```
class SpiTransaction {
    uint8_t csPin;
public:
    explicit SpiTransaction(uint8_t cs,
                            uint32_t freq   = 4'000'000,
                            uint8_t  bitOrd = MSBFIRST,
                            uint8_t  mode   = SPI_MODE0)
        : csPin(cs) {
        SPI.beginTransaction(SPISettings(freq, bitOrd, mode));
        digitalWrite(csPin, LOW);    // aktivace chip select
    }

    ~SpiTransaction() {
        digitalWrite(csPin, HIGH);   // deaktivace
        SPI.endTransaction();        // vždy, i při výjimce
    }

    SpiTransaction(const SpiTransaction&)            = delete;
    SpiTransaction& operator=(const SpiTransaction&) = delete;

    uint8_t transfer(uint8_t b)             { return SPI.transfer(b); }
    void    transfer(uint8_t* buf, size_t n){ SPI.transfer(buf, n);   }
};
```

RAII — SPI transakce jako guard (2/2)

Použití je čisté — na `endTransaction` vůbec nemusíme myslet:

```
void sendPacket(uint8_t* data, size_t len) {
    SpiTransaction txn(CS_PIN, 8'000'000);   // CS LOW, beginTransaction
    txn.transfer(data, len);
}   // ← CS HIGH, endTransaction — automaticky při opuštění scope
```

RAII — kritická sekce (FreeRTOS / ESP32)

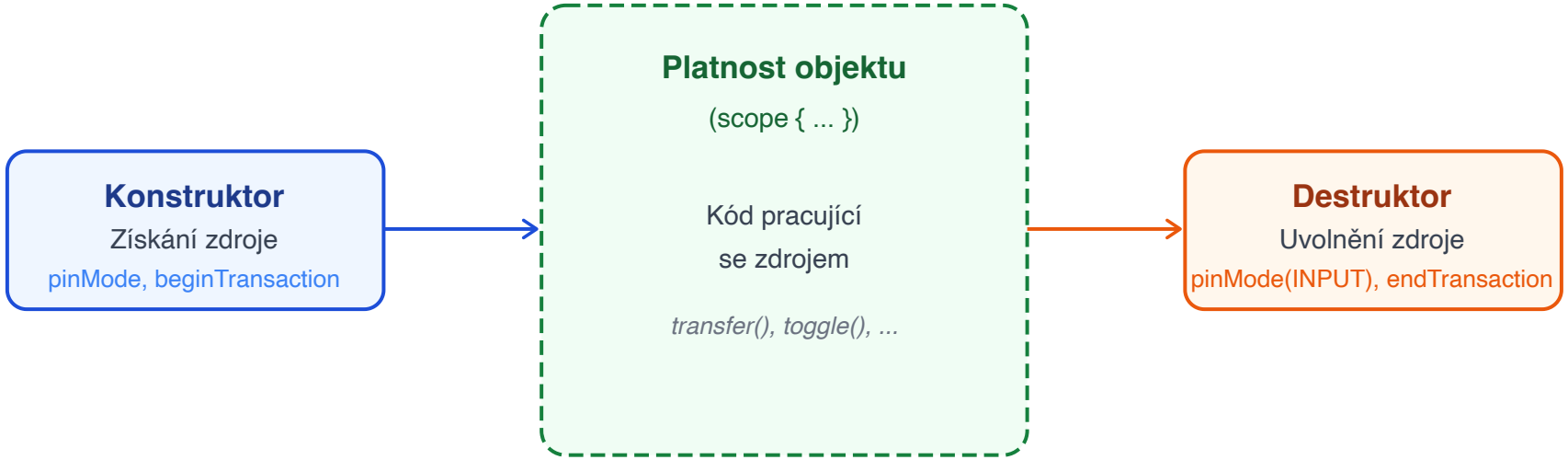
Na systémech s RTOS nebo přerušeními je kritická sekce RAII klasikou. Bez ní bychom museli ručně volat `taskENTER_CRITICAL()` a `taskEXIT_CRITICAL()` — a nesměli by jsme zapomenout ani v jednom return path.

```
class CritickaSecce {
public:
    CritickaSecce()  { taskENTER_CRITICAL(); }   // zakáže přerušení
    ~CritickaSecce() { taskEXIT_CRITICAL(); }    // obnoví přerušení

    CritickaSecce(const CritickaSecce&)            = delete;
    CritickaSecce& operator=(const CritickaSecce&) = delete;
};

// Použití – kritická sekce platí po dobu života objektu
void aktualizujSdilenaData() {
    CritickaSecce cs;           // přerušení zakázána od tohoto bodu

    sdilenyBuffer[idx] = noveData;
    idx = (idx + 1) % BUF_SIZE;

}   // ← přerušení obnovena automaticky, i kdyby funkce hodila výjimku
```

Všimněte si, že nemusíme mít `try/finally` ani se starat o `return` uprostřed funkce. Destruktor se zavolá vždy.

> Na bare-metal systémech bez RTOS nahradíme `taskENTER_CRITICAL()` voláním `noInterrupts()` (Arduino) nebo přímo zápisem do PRIMASK registru na ARM Cortex-M.

RAII — shrnutí vzorů pro embedded



Destruktor se zavolá automaticky — i při výjimce, i při předčasném return

Zdroj	Získání (konstruktor)	Uvolnění (destruktor)
GPIO pin	`pinMode(OUTPUT)`	`pinMode(INPUT)`
SPI transakce	`beginTransaction`, CS LOW	CS HIGH, `endTransaction`
I2C transakce	`Wire.beginTransmission`	`Wire.endTransmission`
Kritická sekce	`noInterrupts()`	`interrupts()`
DMA kanál	`startDma()`	`stopDma()`, čekání na dokončení

Šablony pro HW abstrakci

Templates

Typově bezpečný, nulově nákladný generický kód

Šablony — připomenutí a motivace (1/2)

Šablony v C++ generují kód při překladu — každá specializace je samostatná funkce nebo třída v binárním výstupu. Na MCU to znamená: žádné nepřímé volání přes ukazatel na funkci, žádný runtime overhead.

Klasický problém: chceme obecný kruhový buffer pro různé typy dat a různé velikosti. V C bychom použili `void*` a ztratili typovou bezpečnost. V C++ máme šablony.

```
// Generický kruhový buffer — typ T a pevná velikost N jsou šablonové parametry
template<typename T, size_t N>
class RingBuffer {
    T      buf[N];      // staticky alokováno — žádný heap!
    size_t head  = 0;
    size_t tail  = 0;
    size_t count = 0;

public:
    // Vložení na konec — vrátí false pokud plný
    bool push(const T& val) {
        if (count == N) return false;
        buf[head] = val;
        head = (head + 1) % N;
        ++count;
        return true;
    }
```

Šablony — připomenutí a motivace (2/2)

Všimněte si, že `buf[N]` je pole pevné velikosti — leží buď na zásobníku (lokální proměnná) nebo v datové sekci (globální/statická). Nikdy na heapu.

```
    // Odebrání ze začátku — vrátí false pokud prázdný
    bool pop(T& out) {
        if (count == 0) return false;
        out  = buf[tail];
        tail = (tail + 1) % N;
        --count;
        return true;
    }

    size_t size()  const { return count; }
    bool   empty() const { return count == 0; }
    bool   full()  const { return count == N; }
};   // konec RingBuffer
```

RingBuffer — použití (1/2)

Šablona se specializuje při použití. Každá kombinace `T` a `N` je samostatná třída — překladač ji vygeneruje při kompilaci.

```
// Specializace 1: buffer pro UART bajty, 256 prvků
RingBuffer<uint8_t, 256> uartRx;

// Specializace 2: buffer pro naměřené hodnoty, 64 prvků
RingBuffer<float, 64> senzorData;

// Specializace 3: buffer pro struktury, 16 prvků
struct Paket { uint8_t id; float data[3]; uint16_t crc; };
RingBuffer<Paket, 16> paketovyBuffer;
```

RingBuffer — použití (2/2)

Všechny tři buffery jsou **typově oddělené** — nelze omylem vložit `float` do `uartRx`. Vše se kontroluje při překladu, žádný runtime overhead.

```
// Typická použití – z ISR:
void IRAM_ATTR uart_rx_isr() {
    uint8_t b = UART0.fifo.rw_byte;
    uartRx.push(b);              // O(1), žádná alokace
}

// Z loop():
void zpracujPrijata() {
    uint8_t b;
    while (uartRx.pop(b)) {
        protokol.zpracuj(b);
    }
}
```

> Velikost `N` je šablonový parametr — buffer leží celý na zásobníku nebo jako globální proměnná, nikdy na heapu. To je zásadní výhoda oproti `std::vector`.

Šablonový parametr pro číslo pinu (1/2)

Jedním z nejelegantnějších způsobů využití šablon v embedded je **číslo pinu jako šablonový parametr**. Pin je znám při překladu — kompilátor může celou manipulaci s pinem vložit inline a optimalizovat na pár instrukcí.

```
template<uint8_t PIN>
class DigPin {
    static_assert(PIN < NUM_DIGITAL_PINS,
                  "Pin number out of range");   // kontrola při překladu!
public:
    static void setupOutput() { pinMode(PIN, OUTPUT); }
    static void setupInput()  { pinMode(PIN, INPUT_PULLUP); }
    static void high()        { digitalWrite(PIN, HIGH); }
    static void low()         { digitalWrite(PIN, LOW); }
    static void toggle()      { digitalWrite(PIN, !digitalRead(PIN)); }
    static bool read()        { return digitalRead(PIN); }
};
```

Šablonový parametr pro číslo pinu (1/2)

Všechny metody jsou `static` — objekt vůbec není potřeba. Pin je zakódován v typu.

```
// Typy, ne hodnoty — záměna typů = chyba překladu
using LedPin  = DigPin<LED_BUILTIN>;
using BtnPin  = DigPin<BOOT_PIN>;
using CS_Pin  = DigPin<5>;

LedPin::setupOutput();
BtnPin::setupInput();

LedPin::high();        // kompilátor zná PIN = 2 při překladu → přímá instrukce
BtnPin::read();        // žádné nepřímé volání, žádný ukazatel na funkci
```

Šablony — typová bezpečnost pro registry (1/2)

Šablony lze použít i pro přístup k registrům — přidávají typovou bezpečnost a pojmenování, bez jakéhokoliv runtime overhead. Adresa registru je šablonovým parametrem — jde do kódu, ne do RAM.

```
// Obecný přístupový objekt pro jeden MMIO registr
template<uint32_t ADDRESS, typename T = uint32_t>
struct Register {
    // Vrátí volatile referenci na danou adresu
    static volatile T& ref() {
        return *reinterpret_cast<volatile T*>(ADDRESS);
    }

    static T    read()          { return ref();        }
    static void write(T val)    { ref() = val;         }
    static void setBit(T mask)  { ref() |= mask;       }
    static void clrBit(T mask)  { ref() &= ~mask;      }
    static void toggle(T mask)  { ref() ^= mask;       }
    static bool testBit(T mask) { return (ref() & mask) != 0; }
};
```

Šablony — typová bezpečnost pro registry (2/2)

Konkrétní registry pojmenujeme pomocí `using` aliasů. Kód pak pracuje se smysluplnými jmény místo surových adres — a překladač zkontroluje, zda pracujeme se správným typem.

```
// Pojmenované registry GPIO portu A (STM32F4, z datasheets)
using GPIOA_MODER = Register<0x4002'0000>;   // Mode register
using GPIOA_ODR   = Register<0x4002'0014>;   // Output data register
using GPIOA_IDR   = Register<0x4002'0010>;   // Input data register

// Použití — čitelné, pojmenované, typově bezpečné, zero overhead:
GPIOA_ODR::setBit(1u << 5);              // pin PA5 → HIGH
GPIOA_ODR::clrBit(1u << 5);              // pin PA5 → LOW
bool btn = GPIOA_IDR::testBit(1u << 0); // čti stav PA0
```

`constexpr` a compile-time výpočty

Zero runtime cost

Co se vypočítá při překladu, nevyžaduje čas za běhu

`constexpr` — základ

`constexpr` říká překladači: *"toto lze (nebo musí) vyhodnotit při překladu."* Výsledkem je konstanta uložená ve Flash (ROM), ne výpočet za běhu.

```
// Bez constexpr — výpočet probíhá za běhu (zbytečně)
const float PI = 3.14159265f;
const float circumference = 2.0f * PI * 5.0f;   // 2 FPU operace za běhu

// S constexpr — výsledek je znám při překladu, uložen jako konstanta
constexpr float PI = 3.14159265f;
constexpr float circumference = 2.0f * PI * 5.0f;   // 0 operací za běhu
```

`constexpr` funkce se vyhodnotí při překladu, pokud jsou argumenty konstanty. Se runtime argumenty funguje jako normální funkce.

```
// Funkce, která funguje v obou módech
constexpr uint32_t bitMask(uint8_t bit) {
    return 1u << bit;
}

constexpr uint32_t LED_MASK = bitMask(5);   // → 0x20, uloženo v ROM
uint32_t dynMask = bitMask(uzivatelskyPin); // → volání funkce za běhu
```

`constexpr` — lookup tabulky

Nejsilnější použití v embedded: předpočítané lookup tabulky. Místo toho, aby MCU počítal sinus nebo log za běhu, najde výsledek v tabulce ve Flash.

```
#include <array>
#include <cmath>

// Generovací funkce — celá proběhne při překladu
constexpr std::array<uint8_t, 256> makeSineTable() {
    std::array<uint8_t, 256> table{};
    for (size_t i = 0; i < 256; ++i) {
        // Mapujeme sinus <-1, 1> na <0, 255> pro PWM
        table[i] = static_cast<uint8_t>(
            127.5f + 127.5f * std::sin(2.0f * 3.14159265f * i / 256.0f)
        );
    }
    return table;
}

// Tabulka 256 bajtů leží ve Flash — výpočet proběhl při kompilaci
constexpr auto SINE_TABLE = makeSineTable();

// Za běhu: jen přístup do pole — žádné FPU operace
uint8_t pwmVal = SINE_TABLE[faze];   // O(1), jeden přístup do ROM
```

Tabulka má 256 bajtů ve Flash a nula bajtů RAM. Na MCU bez FPU (ATmega) by výpočet sinu za běhu trval stovky cyklů a spoustu RAM.

`constexpr` — konfigurace a pin mapping

`constexpr` se skvěle hodí pro konfiguraci hardware — pin mapping, frekvence, časové konstanty. Vše je kontrolováno a vyhodnocováno při překladu.

```
// Konfigurace desky — vše constexpr, žádná RAM
struct BoardConfig {
    static constexpr uint8_t  LED_PIN     = 2;
    static constexpr uint8_t  SPI_CS_PIN  = 5;
    static constexpr uint8_t  SPI_CLK_PIN = 18;
    static constexpr uint32_t SPI_FREQ_HZ = 8'000'000;
    static constexpr uint8_t  I2C_SDA_PIN = 21;
    static constexpr uint8_t  I2C_SCL_PIN = 22;
    static constexpr uint32_t I2C_FREQ_HZ = 400'000;   // fast mode
    static constexpr uint32_t UART_BAUD   = 115'200;
};
```

Frekvence a časové konstanty lze z konfigurace odvozovat dalšími `constexpr` funkcemi:

```
// Výpočet počtu cyklů timeru při překladu
constexpr uint32_t msToTicks(uint32_t ms, uint32_t cpuFreq = 240'000'000) {
    return ms * (cpuFreq / 1000);
}

constexpr uint32_t TIMEOUT_TICKS  = msToTicks(500);   // 500 ms
constexpr uint32_t DEBOUNCE_TICKS = msToTicks(20);    //  20 ms debounce

// Výpočet děliče UART baud rate (zaokrouhlení při překladu)
constexpr uint32_t calcBaudrateDiv(uint32_t baud, uint32_t pclk) {
    return (pclk + baud / 2) / baud;
}
constexpr uint32_t USART_BRR = calcBaudrateDiv(115'200, 42'000'000);

constexpr uint32_t msToTicks(uint32_t ms, uint32_t cpuFreq = 240'000'000) {
    return ms * (cpuFreq / 1000);
}

constexpr uint32_t TIMEOUT_TICKS  = msToTicks(500);    // 500 ms → ticky
constexpr uint32_t DEBOUNCE_TICKS = msToTicks(20);     //  20 ms debounce

// Přenosové rychlosti
constexpr uint32_t calcBaudrateDiv(uint32_t baud, uint32_t pclk) {
    return (pclk + baud / 2) / baud;   // zaokrouhlení při překladu
}
constexpr uint32_t USART_BRR = calcBaudrateDiv(115200, 42'000'000);
```

STL v embedded

Co použít, co vynechat

Standardní knihovna je velká — ne vše se hodí na MCU

STL — přehled použitelnosti

Standardní knihovna C++ je navržena pro systémy s neomezenou pamětí a výjimkami. Na MCU potřebujeme vybírat.

Doporučeno

- std::array<T, N>
- std::span<T> (C++20)
- std::optional<T>
- std::variant<...>
- std::string_view
- <algorithm> (sort, find…)
- <numeric> (accumulate…)
- <cmath> (sin, cos…)
- std::bitset<N>

S rozvahou

- std::function<>
  heap alokace pro capture
- std::tuple
- std::initializer_list
- <chrono> typy
- std::pair
- std::unique_ptr
  OK pokud heap je povolený

Vyhněte se

- std::vector, std::list
  dynamická alokace
- std::map, std::set
  heap + velký overhead
- std::string
  heap alokace
- std::iostream
  velký kódový overhead
- výjimky (throw/catch)

`std::array` — náhrada za C pole

`std::array<T, N>` je přímá náhrada za C pole `T[N]`. Na MCU je to zásadní zlepšení: zná svou velikost, lze ho předávat hodnotou nebo referencí, funguje s algoritmy STL, a **neprovádí žádnou alokaci**.

```
#include <array>
#include <algorithm>
#include <numeric>

// C pole — nebezpečné, nezná vlastní velikost
uint8_t buf_c[64];
processData(buf_c, 64);    // musíme předávat velikost zvlášť

// std::array — bezpečné, zná velikost
std::array<uint8_t, 64> buf;
buf.fill(0);               // nuluj celý buffer
buf.size();                // → 64, bez parametru

// Funguje s algoritmy STL — najdi maximum
auto maxVal = *std::max_element(buf.begin(), buf.end());

// Funguje s range-for
for (uint8_t& b : buf) { b = readSensor(); }

// Součet přes std::accumulate
uint32_t soucet = std::accumulate(buf.begin(), buf.end(), 0u);
```

Velikost bufferu je šablonový parametr — překladač zná ji při překladu a může lépe optimalizovat.

`std::span` — bezpečné předávání bufferů

`std::span<T>` (C++20) řeší problém předávání polí funkcím. Nahrazuje dvojici `(T* ptr, size_t len)` jedním typem, který nese oba informace a zachovává typovou bezpečnost.

```
#include <span>

// Stará C++ funkce – nebezpečná
void zpracuj(uint8_t* data, size_t len) { /* ... */ }

// Moderní – std::span, nulový overhead, nevlastní paměť
void zpracuj(std::span<uint8_t> data) {
    for (uint8_t& b : data) { b ^= 0xFF; }   // invertuj bity
}

// Funguje s čímkoliv – std::array, C pole, část pole
std::array<uint8_t, 64> arr;
uint8_t raw[64];

zpracuj(arr);              // celý std::array
zpracuj(raw);              // celé C pole – detekuje velikost!
zpracuj({raw + 10, 20});  // 20 prvků začínající od indexu 10

// Podspan – bez kopírování
auto prvnichDeset = std::span<uint8_t>(arr).first(10);
auto poslednich   = std::span<uint8_t>(arr).last(5);
```

`std::span` nevlastní data — je to pouhý pohled (view) na existující buffer. Vhodné pro předávání dat mezi vrstvami ovladače bez kopírování.

`std::optional` — výsledek nebo chyba bez výjimky (1/2)

Na embedded systémech typicky zakážeme výjimky (`-fno-exceptions`) kvůli overhead a nepředvídatelným latencím. `std::optional<T>` nabízí elegantní alternativu pro funkce, které mohou selhat.

```
#include <optional>

// Funkce vracející optional – buď hodnotu, nebo "nic"
std::optional<float> zmerTeplotu(uint8_t addr) {
    if (!wire.beginTransmission(addr)) {
        return std::nullopt;    // senzor nedostupný
    }
    uint8_t data[2];
    wire.readBytes(data, 2);
    float temp = ((data[0] << 8) | data[1]) * 0.0625f;
    return temp;                // úspěch
}
```

`std::optional` — výsledek nebo chyba bez výjimky (2/2)

Volající musí explicitně ošetřit případ selhání — na rozdíl od výjimky, která může propadnout neošetřená.

```
auto t = zmerTeplotu(0x48);
if (t.has_value()) {
    Serial.println(t.value());   // nebo jen *t
} else {
    Serial.println("Senzor nedostupný");
}

// Kratší varianta s value_or()
float teplota = zmerTeplotu(0x48).value_or(-99.0f);
```

> `std::optional` přidá k objektu jen jeden bajt (příznak platnosti). Na Arduinu / ESP32 s C++17 je plně dostupný.

Proč se vyhýbat dynamické alokaci

`new`, `delete`, `std::vector`, `std::string` — vše pracuje s heapem. Na MCU je to problém hned z několika důvodů:

Fragmentace paměti. Malý heap (typicky 8–200 kB) se po opakované alokaci a uvolňování rozdrobí. Po hodinách provozu může `new` selhat, i když „volné" paměti zdánlivě dost je.

```
Start:  [──────────────── 200 kB volné ────────────────]
Po čase:[██ 4kB ─── 2kB ████ 8kB ── 4kB ████ ── 6kB]
                    ^volné bloky, ale největší = 8kB
```

Nepředvídatelná latence. `malloc` může trvat různě dlouho — závisí na stavu heapu. V real-time systému to naruší časování.

Stack-heap kolize. Zásobník roste dolů, heap roste nahoru. Pokud se potkají, firmware havaruje bez jakéhokoliv varování.

Bezpečná alternativa: statická alokace s pevnou maximální velikostí

```
template<typename T, size_t MAX>
class StaticVector {
    std::array<T, MAX> data;
    size_t             count = 0;
public:
    bool push_back(const T& v) {
        if (count >= MAX) return false;   // selhání je explicitní
        data[count++] = v;
        return true;
    }
    T&     operator[](size_t i) { return data[i]; }
    size_t size() const         { return count; }
};

StaticVector<Paket, 32> fronta;   // max 32 paketů, žádný heap
```

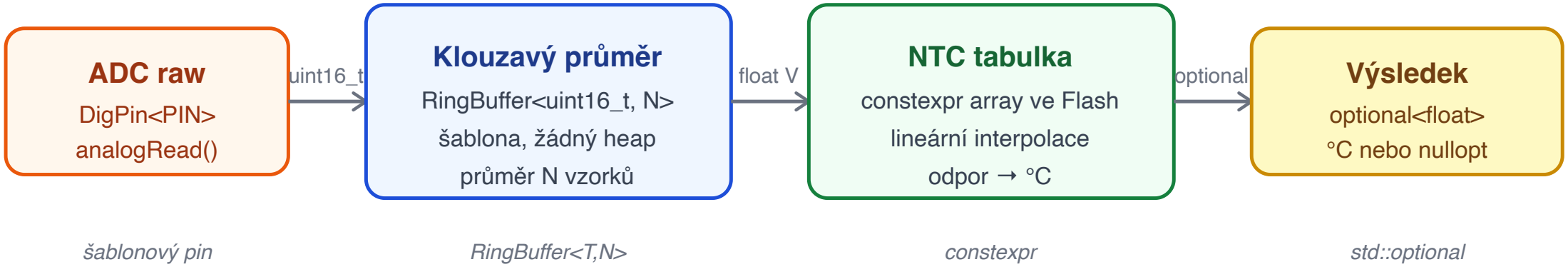
Praktický příklad

Typově bezpečný driver pro SSD1306

Syntéza: šablony + RAII + constexpr + STL bez heap

NTC termistor + klouzavý průměr — přehled

NTC termistor mění odpor s teplotou nelineárně. Přečteme napětí z ADC, filtrujeme klouzavým průměrem a teplotu vyhledáme v `constexpr` tabulce ve Flash. Celá pipeline v jedné třídě.



Každý krok pipeline odpovídá jedné technice z přednášky. Výsledek je `std::optional<float>` — volající musí explicitně ošetřit případ, kdy je hodnota mimo rozsah tabulky.

NTC — `constexpr` kalibrační tabulka

NTC termistor 10 kΩ (B = 3950 K) má nelineární charakteristiku. Tabulku spárovacích hodnot napětí → teplota předpočítáme při překladu a uložíme do Flash.

```
#include <array>
#include <optional>
#include <cstdint>

struct NtcBod {     // Jeden bod kalibrační tabulky
    uint16_t adc;   // ADC hodnota (0–4095 pro 12bit, 0–1023 pro 10bit)
    int8_t   temp;  // teplota v °C (rozsah -128..+127 stačí)
};

// Tabulka naměřená nebo vypočítaná z B-parametrické rovnice
// Uložena ve Flash – žádná RAM. Hodnoty pro NTC 10kΩ, Vcc=3.3V, R_ref=10kΩ
constexpr std::array<NtcBod, 9> NTC_TABLE = {{
    {3900, -20},   // ADC ~3900 → -20 °C (vysoký odpor = chladný)
    {3650, -10},
    {3320,   0},
    {2900,  10},
    {2480,  20},   // při pokojové teplotě ≈ 2480
    {2060,  30},
    {1670,  40},
    {1320,  50},
    {1020,  60},   // ADC ~1020 → +60 °C (nízký odpor = teplý)
}};
```

Tabulka zabírá 18 bajtů ve Flash a 0 bajtů RAM. Rozsah lze snadno rozšířit přidáním dalších bodů — interpolace bude fungovat automaticky.

NTC — lineární interpolace a `std::optional`

Mezi body tabulky interpolujeme lineárně. Výsledek je `std::optional<float>` — pokud je ADC hodnota mimo rozsah tabulky, vrátíme `nullopt` místo nesmyslného čísla.

```
// Interpolace v tabulce — hledáme dva sousední body
std::optional<float> adcNaTeplotu(uint16_t adc) {
    // Mimo rozsah → není smysluplný výsledek
    if (adc > NTC_TABLE.front().adc) return std::nullopt;  // příliš studený
    if (adc < NTC_TABLE.back().adc)  return std::nullopt;  // příliš teplý

    // Najdi první bod s hodnotou adc <= zadané (tabulka je sestupná)
    for (size_t i = 1; i < NTC_TABLE.size(); ++i) {
        if (adc >= NTC_TABLE[i].adc) {
            // Lineární interpolace mezi body [i-1] a [i]
            const auto& hi = NTC_TABLE[i - 1];   // vyšší ADC = nižší temp
            const auto& lo = NTC_TABLE[i];        // nižší ADC = vyšší temp

            float t = (float)(adc - hi.adc) / (lo.adc - hi.adc);
            return hi.temp + t * (lo.temp - hi.temp);
        }
    }
    return std::nullopt;
}
```

Lineární interpolace mezi dvěma body tabulky je velmi přesná, pokud jsou body dostatečně blízko. Pro NTC s krokem 10 °C je chyba interpolace typicky pod 0,5 °C.

NTC — třída `NtcSenzor` (1/4)

Vše spojíme do jedné třídy. Šablonové parametry `PIN` a `N` jsou určeny při překladu — žádný runtime overhead, žádný heap.

```
template<uint8_t PIN, size_t N = 8>
class NtcSenzor {
    RingBuffer<uint16_t, N> buffer;   // staticky alokováno, N × 2 B

public:
    // Přečti nový vzorek z ADC a vlož ho do klouzavého okna
    void sample() {
        buffer.push(analogRead(PIN));   // šablonový PIN → přímá instrukce
    }
```

NTC — třída `NtcSenzor` (2/4)

`analogRead(PIN)` vrátí 10- nebo 12-bitové celé číslo. Ukládáme ho jako `uint16_t` — nepotřebujeme float dokud nevyvoláme interpolaci. Tím šetříme výpočetní čas v `sample()`, který může být volán z časovačového přerušení.

```
    // Vrátí teplotu nebo nullopt (buffer prázdný / mimo rozsah tabulky)
    std::optional<float> teplota() const {
        if (buffer.empty()) return std::nullopt;

        // Průměr přes celé okno — pracujeme s kopií bufferu
        uint32_t soucet = 0;
        auto     tmp    = buffer;
        uint16_t vzorek;
        size_t   pocet  = 0;
        while (tmp.pop(vzorek)) { soucet += vzorek; ++pocet; }

        uint16_t prumer = static_cast<uint16_t>(soucet / pocet);
        return adcNaTeplotu(prumer);   // lineární interpolace v tabulce
    }
};
```

NTC — třída `NtcSenzor` (3/4) a použití

`NtcSenzor<34, 16>` alokuje buffer 16 × 2 B = 32 B staticky v datové sekci. Typ senzoru (pin, velikost okna) je zakódován v typu — záměna pinů je chyba překladu.

```
NtcSenzor<34, 16> termistor;   // pin 34, okno 16 vzorků

void setup() {
    Serial.begin(115200);
    // Zahřátí bufferu — 16 rychlých čtení při startu
    for (int i = 0; i < 16; ++i) { termistor.sample(); delay(5); }
}

void loop() {
    termistor.sample();   // posune okno: vypadne nejstarší, přijde nový

    auto t = termistor.teplota();   // optional<float>

    if (t) {
        Serial.printf("Teplota: %.1f C\n", *t);
    } else {
        Serial.println("Mimo rozsah!");
    }

    delay(100);   // 10 Hz vzorkování
}
```

NTC — třída `NtcSenzor` (4/4)

Kdybychom chtěli druhý senzor na jiném pinu s jiným oknem — jen nová instance, nulový overhead:

```
NtcSenzor<35, 4>  rychlyTermistor;   // pin 35, rychlé okno 4 vzorky
NtcSenzor<34, 32> pomályTermistor;   // pin 34, silné vyhlazení 32 vzorků
```

Celá pipeline — `NtcSenzor` + kalibrační tabulka + interpolace — je **~60 řádků C++, 0 B heap, data tabulky ve Flash**.

Polymorfismus v embedded

Virtual vs. CRTP

Kdy za runtime polymorfismus platíme a kdy ne

Cena virtuálních funkcí (1/2)

Virtuální funkce jsou mocný nástroj — ale mají konkrétní cenu, která na MCU může vadit.

Každá třída s alespoň jednou virtuální metodou nese **vtable pointer** — skrytý ukazatel uložený v každé instanci třídy.

```
class Senzor {
public:
    virtual float read() = 0;   // virtuální
};

class NtcSenzor : public Senzor {
    float read() override { return adcNaTeplotu(analogRead(34)); }
};

// sizeof(NtcSenzor) = sizeof(float) + sizeof(void*)  — 4 nebo 8 B navíc
// Každé volání read() → load vtable pointer → nepřímý skok (branch miss!)
```

Cena virtuálních funkcí (2/2)

Na MCU to bolí ve třech situacích:

- **Těsná smyčka vzorkování** — 100 kHz ADC, každé volání přes vtable = ~3–5 cyklů navíc
- **ISR** — přerušení musí skončit rychle; nepřímé volání narušuje branch predictor
- **Mikrokontroléry bez cache** (ATmega, Cortex-M0) — každý přístup do Flash stojí wait states

Na druhé straně: v kódu volaném jednou za sekundu nebo ve struktuře driverů inicializované při startu je `virtual` naprosto v pořádku a správnou volbou.

Kdy `virtual` dává smysl

Jsou situace, kde runtime polymorfismus je přesně správná volba — a snaha ho nahradit šablonou by kód zkomplikovala bez užitku.

Plugin architektura driverů — driver se vybírá za běhu podle detekovaného hardware nebo konfigurace z EEPROM:

```
// Abstraktní rozhraní senzoru
class ISenzor {
public:
    virtual ~ISenzor()     = default;
    virtual float read()   = 0;
    virtual bool  isOk()   = 0;
};

// Za běhu vybereme správnou implementaci
std::unique_ptr<ISenzor> vytvorSenzor(uint8_t typ) {
    switch (typ) {
        case 0x01: return std::make_unique<DHT22Senzor>(22);
        case 0x02: return std::make_unique<DS18B20Senzor>(23);
        default:   return std::make_unique<DummySenzor>();
    }
}

// Klientský kód neví, co za senzor dostane – a nepotřebuje to vědět
ISenzor* s = vytvorSenzor(eeprom.read(SENSOR_TYPE));
float t = s->read();
```

Dispatch příkazů přes UART / MQTT — přijatý příkaz se přeloží na objekt, který ho zpracuje. Počet příkazů roste za běhu programu, ne při překladu — šablona by sem nesedla.

> Pravidlo palce: `virtual` tam, kde typ objektu závisí na datech nebo konfiguraci za běhu. Šablona tam, kde typ je znám při překladu.

CRTP — polymorfismus při překladu

CRTP (Curiously Recurring Template Pattern) je technika, která dosáhne polymorfního rozhraní **bez vtable** — přeložení se odehraje při kompilaci.

Základní myšlenka: základní třída dostane jako šablonový parametr svou vlastní podtřídu a volá její metody přes `static_cast`.

```
// Základní třída — šablonový parametr Derived je podtřída sama
template<typename Derived>
class SenzorBase {
public:
    // Tato metoda je sdílená pro všechny senzory
    float readFiltered() {
        float soucet = 0;
        for (int i = 0; i < 8; ++i)
            soucet += impl().read();    // volá podtřídu — bez vtable!
        return soucet / 8.0f;
    }

    // Logování — opět sdílené, ale volá podtřídní metodu
    void logValue() {
        Serial.printf("[%s] %.2f\n", impl().name(), impl().read());
    }

private:
    // Bezpečný downcast — Derived je vždy přímá podtřída
    Derived& impl() { return static_cast<Derived&>(*this); }
};
```

CRTP — konkrétní senzory (1/2)

Každý senzor dědí ze `SenzorBase<SamSebe>` a implementuje jen metody specifické pro daný hardware.

```
// NTC termistor jako CRTP senzor
class NtcDriver : public SenzorBase<NtcDriver> {
public:
    float read()        { return adcNaTeplotu(analogRead(34)).value_or(-99); }
    const char* name()  { return "NTC"; }
};

// DHT22 jako CRTP senzor
class DhtDriver : public SenzorBase<DhtDriver> {
public:
    float read()        { return dht.readTemperature(); }
    const char* name()  { return "DHT22"; }
};
```

CRTP — konkrétní senzory (2/2)

Sdílená logika (filtrování, logování) se volá přes `SenzorBase` — ale vždy konkrétní implementací, bez runtime rozhodování:

```
NtcDriver ntc;
DhtDriver dht;

// readFiltered() a logValue() jsou sdílené z SenzorBase,
// ale volají NtcDriver::read() resp. DhtDriver::read() — inline, bez vtable:
float t1 = ntc.readFiltered();   // 8× NtcDriver::read(), pak průměr
float t2 = dht.readFiltered();   // 8× DhtDriver::read(), pak průměr
ntc.logValue();                  // "[NTC] 23.50"
```

Překladač vygeneruje **dvě různé** verze `readFiltered()` — jednu pro NTC, jednu pro DHT22. Žádné vtable, žádné nepřímé volání, výsledek je shodný s ručně psanou funkcí.

Virtual vs. CRTP — srovnání

Aspekt	`virtual`	CRTP
Výběr implementace	za běhu	při překladu
Vtable pointer	ano (+4–8 B/objekt)	ne
Volání metody	nepřímé (vtable lookup)	přímé / inline
Různé typy v jednom kontejneru	`vector<ISenzor*>` ✓	✗ nelze
Typ závisí na datech/konfiguraci	✓	✗
Vhodné pro hot path / ISR	opatrně	✓ ideální
Složitost kódu	nízká	střední

```
// Virtual — různé typy v jednom poli (nelze s CRTP)
std::array<ISenzor*, 4> senzory = { &ntc, &dht, &bme, &ds18 };
for (auto* s : senzory) Serial.println(s->read());

// CRTP — homogenní pole stejného typu, ale zero-cost
std::array<NtcDriver, 4> ntcPole;
for (auto& s : ntcPole) Serial.println(s.readFiltered());
```

Zlaté pravidlo: začněte s `virtual` — je čitelnější. Přejděte na CRTP pouze tehdy, kdy profilerem prokážete, že vtable je měřitelný bottleneck.

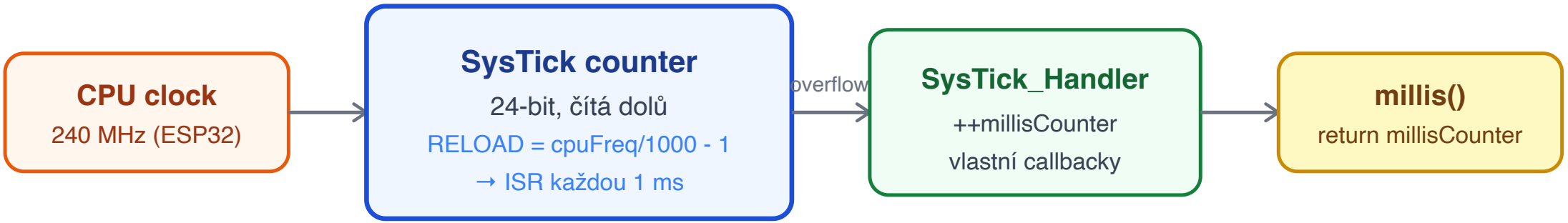
SysTick

Systémový časovač ARM Cortex-M

`millis()` *pod pokličkou a RAII timeout guard*

Co je SysTick

SysTick je 24-bitový countdown časovač zabudovaný přímo do jádra ARM Cortex-M. Čítá dolů od hodnoty `RELOAD` na nulu, pak generuje přerušení a začíná znovu. Arduino ho používá pro `millis()` a `delay()` — ale to nemusí být konec.



SysTick_BASE: 0xE000E010 — pevná adresa ve všech Cortex-M

Čtyři MMIO registry řídí celý SysTick. Leží na pevné adrese `0xE000E010` — stejné ve všech ARM Cortex-M čipech bez ohledu na výrobce.

SysTick — registry přes MMIO

Definujeme přístup k registrům pomocí technik z přednášky — `constexpr` adresy, šablonový `Register<>`.

```
// Pevné adresy SysTick registrů (ARMv7-M Architecture Reference Manual)
constexpr uint32_t SYSTICK_BASE   = 0xE000'E010;

// Jednotlivé registry jako pojmenované typy
using SysTick_CTRL  = Register<SYSTICK_BASE + 0x00>;  // Control & Status
using SysTick_LOAD  = Register<SYSTICK_BASE + 0x04>;  // Reload hodnota
using SysTick_VAL   = Register<SYSTICK_BASE + 0x08>;  // Aktuální hodnota
using SysTick_CALIB = Register<SYSTICK_BASE + 0x0C>;  // Kalibrace

// Bitové masky CTRL registru
constexpr uint32_t CTRL_ENABLE    = 1u << 0;   // spusť časovač
constexpr uint32_t CTRL_TICKINT   = 1u << 1;   // povol přerušení
constexpr uint32_t CTRL_CLKSRC    = 1u << 2;   // zdroj: CPU clock
constexpr uint32_t CTRL_COUNTFLAG = 1u << 16;  // přetekl od posledního čtení?
```

Inicializace SysTick pro 1 ms tik při 240 MHz (ESP32):

```
constexpr uint32_t CPU_FREQ_HZ = 240'000'000;
constexpr uint32_t TICK_HZ     = 1'000;          // 1 ms = 1000 Hz

void systickInit() {
    SysTick_LOAD::write(CPU_FREQ_HZ / TICK_HZ - 1);  // RELOAD hodnota
    SysTick_VAL::write(0);                             // reset čítače
    SysTick_CTRL::write(CTRL_ENABLE | CTRL_TICKINT | CTRL_CLKSRC);
}
```

SysTick — ISR a `volatile` counter

Přerušení `SysTick_Handler` se volá každou 1 ms. Je to přesně ta situace, kdy potřebujeme `volatile` — proměnná `sysMillis` se mění mimo normální tok programu.

```
// volatile – čtení z loop() musí vždy sáhnout do RAM, ne do registru CPU
volatile uint32_t sysMillis = 0;

// Jméno funkce je pevné – linker ho napojí na vektor přerušení
extern "C" void SysTick_Handler() {
    ++sysMillis;            // atomická operace na 32-bit Cortex-M
    // Zde lze volat krátké callbacky – debounce, software timery...
}

// Bezpečné čtení z hlavního kódu
uint32_t millis() {
    return sysMillis;       // volatile zaručí čtení z RAM
}

uint32_t micros() {
    // Kombinujeme millisCounter s aktuální hodnotou čítače
    uint32_t ms  = sysMillis;
    uint32_t cnt = SysTick_VAL::read();               // čítá dolů
    uint32_t reload = SysTick_LOAD::read();
    return ms * 1000 + (reload - cnt) / (CPU_FREQ_HZ / 1'000'000);
}
```

`micros()` ukazuje sílu přímého přístupu k SysTick registrům — Arduino API tuto funkci implementuje velmi podobně, ale skrývá ji za makra a `#ifdef`.

SysTick — RAII `Timeout` guard

Přímý přístup k `millis()` vede k rozptýleným `if (millis() - start > timeout)` podmínkám po celém kódu. RAII guard to centralizuje.

```
class Timeout {
    uint32_t deadline;
public:
    // Konstruktor: zapamatuj si čas vypršení
    explicit Timeout(uint32_t ms) : deadline(millis() + ms) {}

    bool expired() const {
        // Správné porovnání i při přetečení uint32_t (~49 dní)
        return static_cast<int32_t>(millis() - deadline) >= 0;
    }

    void reset(uint32_t ms) { deadline = millis() + ms; }

    bool waitFor(uint32_t ms, auto condition) { // Čekání s možností záchranné podmínky
        Timeout inner(ms);
        while (!inner.expired()) {
            if (condition()) return true;
        }
        return false;   // vypršelo bez splnění podmínky
    }
};
```

Timeout s přetečením je klasická past — `millis() - start > 5000` selže při přetečení `uint32_t`. `int32_t` cast v `expired()` tento případ ošetřuje správně.

SysTick — `Timeout` v praxi

```
bool inicializujSenzor(uint8_t addr) { // Čekání na odpověď senzoru s timeoutem — čitelné a bezpečné
    Wire.beginTransmission(addr);
    Wire.write(CMD_RESET);
    Wire.endTransmission();

    Timeout t(100); // Čekáme max 100 ms na odpověď — RAII, žádný globální stav
    while (!t.expired()) {
        if (senzorOdpovida(addr)) return true;
        delay(1);
    }
    return false;   // timeout
}

void komunikacniSmycka() {          // Různé timeouty pro různé fáze — přehledné
    Timeout celkovy(5000);          // celá operace max 5 s
    Timeout jedenPaket(200);        // jeden paket max 200 ms

    while (!celkovy.expired()) {
        if (jedenPaket.expired()) {
            odesliPing();
            jedenPaket.reset(200);
        }
        zpracujPrijata();
    }
    Serial.println("Relace ukončena.");
}
```

`Timeout` objekty jsou na zásobníku — 4 B každý (jeden `uint32_t`). Žádná alokace, žádný globální stav, přesně RAII filosofie.

Přehled — techniky a kde je použít

Technika	Použití	Výhoda
`volatile`	MMIO registry, ISR sdílené proměnné	Správné chování, přístup nelze optimalizovat pryč
RAII	GPIO, SPI/I2C transakce, kritická sekce, Timeout	Automatické uklizení i při chybě
Šablony	Kruhový buffer, typový pin, MMIO registr	Typová bezpečnost, zero-cost
CRTP	Sdílená logika bez vtable — hot path, ISR	Polymorfismus při překladu, inline
`constexpr`	Lookup tabulky, pin mapping, init sekvence	Výpočet při překladu, data ve Flash
`std::array`	Buffery, framebuffer, font	Zná velikost, žádný heap
`std::span`	Předávání bufferů funkcím	Nulový overhead, bezpečné
`std::optional`	Funkce, které mohou selhat	Bez výjimek, explicitní ošetření
Statická alokace	Vše co by bylo `vector` / `string`	Předvídatelná paměť, žádná fragmentace
SysTick MMIO	Přesné časování, `micros()`, Timeout	Přímý přístup k HW, žádná závislost na SDK

Doporučená pravidla pro embedded C++

Paměť

- Zakazujeme dynamickou alokaci za běhu — `new` / `delete` použijeme jen v `setup()` / inicializaci
- Velikosti bufferů jsou šablonové parametry nebo `constexpr` — vždy zná je překladač
- Vše co může být `const` nebo `constexpr`, nechť je — data půjdou do Flash místo RAM

Bezpečnost

- RAII pro každý HW zdroj — žádné přímé `begin/end` bez RAII wrapperu
- `[[nodiscard]]` na funkce vracející chybový kód — nelze omylem ignorovat
  - C++ atribut (od C++17), který říká překladači: "návratová hodnota této funkce nesmí být ignorována."
  - Pokud volající výsledek zahodí, překladač vydá varování.
- `static_assert` pro kontroly při překladu — čísla pinů, velikosti bufferů

Výkon

- `inline` a `constexpr` pro funkce volané v ISR nebo tight loop
- `IRAM_ATTR` (ESP32) / `__attribute__((section(".itcm")))` pro kód citlivý na latenci
- Vyhněte se `std::function` v hot path — skrytá heap alokace při capture

```
// Dobré návyky v jednom příkladu:
template<uint8_t PIN>
class Button {
    static_assert(PIN < NUM_DIGITAL_PINS, "Invalid pin");
public:
    [[nodiscard]] bool isPressed() const {
        return !digitalRead(PIN);   // active low
    }
};
```

Shrnutí

Moderní C++ na MCU není kompromis — je to **upgrade bez ceny za běhu**.

`volatile` · **RAII** · **šablony** · **CRTP** · `constexpr` · `std::array/span/optional` · **SysTick**

> *"C++ is a language for systems programming.*
> *It gives you full control — but the good kind,*
> *where you don't pay for what you don't use."*
> *— Herb Sutter*

```
// Cíl: kód čitelný jako Python, rychlý jako C, bezpečný jako Rust (téměř)

NtcSenzor<34, 16>    termistor;       // šablona + RingBuffer, žádný heap
SpiTransaction       txn(CS_PIN);     // RAII — CS a endTransaction automaticky
Timeout              t(500);          // SysTick + RAII
constexpr auto       SINE = makeSineTable();   // 256 B ve Flash, 0 B RAM

template<typename D> class SenzorBase { /* CRTP — sdílená logika bez vtable */ };
```