Modern C++ for Embedded Systems

From Fundamentals to Real-Time Solutions

RUTVIJ GIRISH KARKHANIS

Contents

Kickstarting With The Basics

- Class Types
- Namespaces
- Constant Expressions
- Static Asserts
- Basic STL Algos
- atomic_load() and atomic_store()
- align_of() and align_as()
- Abstraction of contiguous memory using

Object-Oriented Techniques For Embedded Systems

- Dynamic Polymorphism
- Cost Of Dynamic Polymorphism
- Non-copyable Classes
- Constant Methods
- Friend Classes And Friend Functions
- Template Functions
- Template Class Type
- CTAD
- Static Polymorphism
- Variadic Templates
- Template Metaprogramming Basics
- Type Traits
- Variable Templates
- SFINAE and enable_if

Introduction *cont.*

Optimizing C++ for Embedded Systems

- Compiler Optimizations
- Using assembly Listings
- Using Map Files
- Name Mangling And De-mangling
- Utilizing ROM efficiently
- Custom Memory Management Using Placement-New
- Lambda Expressions
- Register Access
- Startup Code in C++
- Static Initialization Considerations
- Multithreading in C++

Reusable Mathematical Utilities In C++

- Floating-Point Math
- Mathematical Constants
- Complex Values
- Fixed-Point Math


JS
Css
<html></html>
KICKSTARTING
WITH THE BASICS

Class types

- Classes are the way data is defined in C++
- Similar to C structures, but a constructor, function members, and access specifiers
- Typically used with object-oriented features like inheritance, encapsulation and polymorphism

```
class Point {
public:
    /**
     * @brief Default constructor. Initializes the point
to the origin (0.0, 0.0)
     * using an initializer list
     */
    constexpr Point() : x(0.0), y(0.0) {}

    /**
     * @brief Constructs a Point with specified
coordinates.
     * @param x_val The initial x-coordinate.
     * @param y_val The initial y-coordinate.
     */
    constexpr Point(double x_val, double y_val) :
x(x_val), y(y_val) {}

    double x; ///< The x-coordinate of the point.
    double y; ///< The y-coordinate of the point.

    /**
     * @brief Calculates the Euclidean distance from the
origin (0,0) to this point.
     * @return The distance as a double.
     */
    double distance_from_origin() const {
        return std::sqrt(x * x + y * y);
    }
};
```

Class types ... *cont.*

- Basic operators can be overloaded to operate on these user-defined types
- The overload function may or may not be member functions of the class

```
/**
 * @brief Overloads the stream insertion operator for easy
printing of a Point.
 * @param os The output stream (e.g., std::cout).
 * @param p The Point object to print.
 * @return A reference to the output stream.
 */
std::ostream& operator<<(std::ostream& os, const Point& p)
{
    os << "(" << p.x << ", " << p.y << ")";
    return os;
}

/**
 * @brief Overloads the addition operator to perform
vector addition on two Points.
 *
 * @param lhs The left-hand side Point.
 * @param rhs The right-hand side Point.
 * @return A new Point representing the sum of the two
points.
 */
constexpr Point operator+(const Point& lhs, const Point&
rhs) {
    return Point(lhs.x + rhs.x, lhs.y + rhs.y);
}
```

Class types ... *cont.*

- Objects of these class types can be initialized in several ways
- The preferred modern style is the Direct-List-Initialization

```
// Default Initialization (Pre-C++11 style)
Point p; // Calls Point::Point()

// Direct Initialization (Pre-C++11 style)
Point p1(3.0, 4.0);  // Calls Point(double x_val, double
y_val) : x(x_val), y(y_val)

// Copy Initialization (Pre-C++11 style)
Point p2 = p1;       // It requires an accessible copy or
move constructor.

// List Initialization (or "Uniform Initialization")
(C++11 and later)
Point p3{5.0, 12.0}; // direct-list-initialization –
Modern approach
p = {2.0, 1.0};      // copy-list-initialization
```

Namespaces

- Group together related symbols
- Contribute to creating unique names for symbols by adding additional naming information
- Used for formalizing the scope

```
// A C++17 nested namespace definition.
namespace A::B::C
{
    Point p;
}

void accessPoint()
{
    A::B::C::p = {2.3, 5.0};
}

// Namespace aliasing for readability
namespace fs = std::filesystem;
```

Constant Expressions

- C++11 introduced the powerful new keyword “constexpr” to define compile-time constants
- The constexpr keyword can make compile-time constants from a wider variety of things than the original “const” keyword
- The main idea of constexpr is to shift the burden of execution from run-time to compile-time

```
// --- Generating a Compile-Time Lookup Table ---
// This pre-computes values and embeds them directly into
the executable,
// avoiding runtime calculation overhead.

/**
 * @brief A namespace for compile-time mathematical
functions.
 */
namespace compile_time_math {
    // A simple constexpr factorial for the Taylor series.
    constexpr long double factorial(int n) {
        return (n <= 1) ? 1.0L : n * factorial(n - 1);
    }

    // A simple constexpr power function.
    constexpr long double power(long double base, int exp)
{
        return (exp == 0) ? 1.0L : base * power(base, exp
- 1);
    }

    // A constexpr implementation of sine using a Taylor
series expansion.
    // Note: std::sin is not generally constexpr before
C++23.
    constexpr double sin_approx(double x) {
        return x
             - power(x, 3) / factorial(3)
             + power(x, 5) / factorial(5)
             - power(x, 7) / factorial(7);
    }
}
```

```
namespace compile_time_math {

    template <std::size_t N>
    constexpr std::array<double, N> make_sine_table() {
        std::array<double, N> table{};
        constexpr double pi = 3.14159265358979323846;
        for (std::size_t i = 0; i < N; ++i) {
            // Map integer index i to an angle in radians [0, 2*pi)
            double angle = 2.0 * pi * i / N;
            table[i] = sin_approx(angle);
        }
        return table;
    }
}

int main() {

    constexpr auto sine_lookup_table =
compile_time_math::make_sine_table<10>();

    return 0;
}
```

```
# Assembly listing for arm7-a
main:
        push    {r11, lr}
        mov     r11, sp
        sub     sp, sp, #96
        ldr     r0, .LCPI0_0
        str     r0, [r11, #-4]
        add     r0, sp, #8
        ldr     r1, .LCPI0_2
.LPC0_0:
        add     r1, pc, r1
        ldr     r2, .LCPI0_3
        bl      memcpy
        vldr    d0, .LCPI0_1
        vstr    d0, [sp]
        ldr     r0, .LCPI0_0
        mov     sp, r11
        pop     {r11, pc}
.LCPI0_1:
        .long   980665216
        .long   1071806924
.LCPI0_0:
        .long   0
.LCPI0_2:
        .long   .L__const.main.sine_lookup_table-
(.LPC0_0+8)
.LCPI0_3:
        .long   80
.L__const.main.sine_lookup_table:
        .long   0
        .long   0
        .long   3949949152
        .long   1071828770
        .long   2088362432
        .long   1072590561
        .long   2631324320
        .long   1072588924
        .long   980665216
        .long   1071806924
        .long   2004507904
        .long   3216196010
        .long   3092404928
        .long   3220096251
        .long   4033698424
        .long   3221426607
        .long   2304970528
        .long   3222663693
        .long   1296545922
        .long   3223975929
```

static_assert

- C++ compiler provides functionality to perform static checks on at compile time Boolean expressions.
- These are particularly useful to avoid bugs such as out-of-range accesses

```
template <std::size_t Index, typename T, std::size_t N>
constexpr const T& get_element_safely(const std::array<T,
N>& arr) {
    // This static_assert ensures the index is within
bounds *at compile time*.
    // If this condition is false, the program will not
compile.
    static_assert(Index < N, "Compile-time bounds check
failed: Index is out of range.");
    return arr[Index];
}

int main() {

    // Create an array
    constexpr std::array<int, 5> my_array = {10, 20, 30,
40, 50};

    // --- Invalid Call (This will cause a COMPILE-TIME
ERROR) ---
    // The following line, if uncommented, will fail to
compile because the
    // static_assert(5 < 5) is false.
    // get_element_safely<5>(my_array); // COMPILE ERROR:
... Index is out of range.
}`
```

Basic STL Algos

- STL consist of some easy-to-use algorithms viz.
  `std::fill()`,
  `std::for_each()`,
  `std::all_of()`,
  `std::inner_product()`

```
std::vector<int> v1(5); // Create a vector of 5 integers
(default-initialized to 0)

std::fill(v1.begin(), v1.end(), 42); // Fill the entire
vector with the value 42

// Determine if all the elements are even
bool all_are_even = std::all_of(v1.cbegin(), v1.cend(),
[](int i){ return i % 2 == 0; });

std::vector<int> v3 = {1, 2, 3, 4};
std::vector<int> v4 = {5, 6, 7, 8};

// The third argument is the initial value of the sum.
// Calculation: 0 + (1*5) + (2*6) + (3*7) + (4*8) = 70
auto dot_product = std::inner_product(v3.cbegin(), v3.cend(),
v4.cbegin(), 0);

// The algorithm is more general. It can take two more
optional arguments:
// an operation for the summation and an operation for the
product.
// Example: init + (v3[0]+v4[0]) + (v3[1]+v4[1]) + ...
// Calculation: 100 + (1+5) + (2+6) + (3+7) + (4+8) = 100 + 6
+ 8 + 10 + 12 = 136
auto sum_of_sums = std::inner_product(v3.cbegin(), v3.cend(),
v4.cbegin(), 100,
                                      std::plus<int>(),
// The "summation" operation
                                      std::plus<int>());
// The "product" operation
```

atomic_load() and atomic_store()

- The <atomic> library in the C++ standard library includes a collection of safe and portable atomic operations
- Here we get a quick start with the std::atomic_load() and the std::atomic_store() functions

```
/// @brief A flag to indicate that new data is available from the
sensor.
/// The ISR sets this to true; the main loop sets it to false after
processing.
std::atomic<bool> g_new_data_available(false);

/// @brief The shared data buffer. In a real system, this might be a
struct of sensor readings.
std::atomic<int> g_sensor_value(0);

void simulate_isr() {
    for (int i = 0; i < 5; ++i) {
        // Simulate regular interrupt.
        std::this_thread::sleep_for(std::chrono::milliseconds(750));

        int new_value = 10 * (i + 1);
        std::cout << "[ISR]   New sensor reading: " << new_value <<
std::endl;

        // --- Producer Side ---
        // 1. Atomically store the new value into the shared variable.
        // This is a single, indivisible operation. The main loop will
never
        // see a "torn" or partially written value. The default memory
order
        // (seq_cst) ensures this write is visible to other threads in
a
        // consistent order.
        std::atomic_store(&g_sensor_value, new_value);

        // 2. Atomically set the flag to signal that data is ready.
        // This write will become visible to the main loop after the
sensor
        // value write is visible.
        std::atomic_store(&g_new_data_available, true);
    }
}
```

atomic_load() and atomic_store() ... *cont.*

```
void main_loop() {
    int processed_count = 0;
    while (processed_count < 5) {
        // --- Consumer Side ---
        // 1. Atomically load the flag to check if new data is
available.
        // This is a safe way to read the flag without data races.
        if (std::atomic_load(&g_new_data_available)) {
            // 2. Atomically load the sensor value for processing.
            int current_value = std::atomic_load(&g_sensor_value);

            // Process the value
            _do_some_work(&current_value);

            // 3. Atomically clear the flag to indicate we are ready
for the next value.
            std::atomic_store(&g_new_data_available, false);

            processed_count++;
        } else {
            // No new data, the main loop can do other things or sleep
("poll").

std::this_thread::sleep_for(std::chrono::milliseconds(50));
        }
    }
}

int main() {
    std::cout << "--- std::atomic for ISR/Main Loop Communication ---"
<< std::endl;

    std::thread isr_thread(simulate_isr);
    main_loop();
    isr_thread.join();

    std::cout << "\nProgram finished." << std::endl;
}
```

alignof() and alignas()

- `alignof(Type)`: An operator that returns the alignment requirement (in bytes) of a given type. This is a compile-time constant.
- `alignas(N)` or `alignas(Type)`: A specifier that enforces a minimum alignment for a variable or struct/class. N must be a power of two.

```
// A struct where we enforce a custom, larger alignment using
alignas.
// This is common for data that will be processed by SIMD
instructions,
// which often require 16, 32, or even 64-byte alignment for
best performance.
struct alignas(32) AlignedForSIMD {
    float data[8]; // A block of 8 floats (32 bytes)
};

// Prints 1
std::cout << "alignof(char): " << alignof(char) << '\n’;

// Prints alignof(AlignedForSIMD): 32 (enforced by
alignas(32))
std::cout << "alignof(AlignedForSIMD): " <<
alignof(AlignedForSIMD) << " (enforced by alignas(32))\n";
```

Abstraction of contiguous memory using

- C++20 provides an addition to the standard library, the library
- std::span represents an efficient abstraction of contiguous memory blocks

```
/**
 * @brief A function that operates on a contiguous sequence of
integers.
 *
 * By accepting a std::span, this function can work with data from a
 * std::vector, a C-style array, a std::array, or a sub-section of any
of them
 * without needing to be templated or causing data to be copied.
 *
 * @param data A non-owning view of some integer data.
 */
void process_data(std::span<const int> data) {
    long long sum = 0;
    std::cout << "Processing a span of " << data.size() << " elements:
[ ";
    for (int val : data) {
        std::cout << val << " ";
        sum += val;
    }
    std::cout << "]. Sum = " << sum << std::endl;
}

int main() {

    // a) With a std::vector
    // Processing a span of 5 elements: [ 1 2 3 4 5 ]. Sum = 15
    std::vector<int> my_vector = {1, 2, 3, 4, 5};
    process_data(my_vector);

    // b) With a C-style array
    // Processing a span of 3 elements: [ 10 20 30 ]. Sum = 60
    int c_array[] = {10, 20, 30};
    process_data(c_array);

    // c) With a std::array
    // Processing a span of 4 elements: [ 100 200 300 400 ]. Sum = 1000
    std::array<int, 4> my_array = {100, 200, 300, 400};
    process_data(my_array);
}
```

OBJECT-ORIENTED TECHNIQUES FOR EMBEDDED SYSTEMS

Concepts, Examples, and Applications

Dynamic Polymorphism

- Dynamic polymorphism makes run-time decisions on specific code execution (function calls)
- Runtime virtual function mechanism is used to call functions of a derived class using a base class pointer or a reference

```
// --- 1. The Abstract Interface ---

class ISensor {
public:
    // A virtual destructor is crucial for any class intended for polymorphic
use.
    // It ensures that the correct derived class destructor is called when an
    // object is deleted through a base class pointer.
    virtual ~ISensor() = default;

    /**
     * @brief Reads a value from the sensor.
     * @return The sensor reading as a float.
     */
    virtual float read() const = 0; // Pure virtual function

    /**
     * @brief Gets the name of the sensor.
     * @return The sensor's name as a string.
     */
    virtual const char* get_name() const = 0; // Pure virtual function
};

// --- 2. Concrete Implementations ---

/**
 * @brief A concrete implementation for a temperature sensor (e.g., a DS18B20).
 */
class TemperatureSensor : public ISensor {
public:
    float read() const override {
        // In a real system, this would involve I2C/SPI/1-Wire communication.
        std::cout << "  (Reading from Temperature Sensor hardware...)\n";
        return 24.5f; // degrees Celsius
    }

    const char* get_name() const override {
        return "TemperatureSensor";
    }
};

/**
 * @brief A concrete implementation for a humidity sensor (e.g., a DHT22).
 */
class HumiditySensor : public ISensor {
public:
    float read() const override {
        // In a real system, this would be different hardware communication.
        std::cout << "  (Reading from Humidity Sensor hardware...)\n";
        return 55.2f; // percent relative humidity
    }

    const char* get_name() const override {
        return "HumiditySensor";
    }
};

// --- 3. The Application Logic ---

std::vector<std::unique_ptr<ISensor>> sensors;
sensors.push_back(std::make_unique<TemperatureSensor>());
sensors.push_back(std::make_unique<HumiditySensor>());

for (const auto& sensor_ptr : sensors) {
    float value = sensor_ptr->read();
}
```

Cost of Dynamic Polymorphism

- The overhead of dynamic polymorphism comes directly from the compiler implementation of the Virtual Function Mechanism

1. **Memory Overhead:**
   There are two components to the memory overhead:

   - **The Virtual Table (vtable)**
     A static array of function pointers that holds the memory addresses of the correct virtual function for that class. This is a *per-class* overhead and is generally very small.
   - **The Virtual Pointer (vptr)**
     This is an extra hidden member variable that points to the vtable of the object's class. This is a *per-object* overhead and has a more significant cost.

2. **Performance Overhead:**
   The performance cost comes from the indirection required to call a virtual function.

   - **Normal Function Call**
     A non-virtual function call is resolved at compile time.
   - **Virtual Function Call**
     A virtual function call can only be resolved at runtime and takes several steps to call the correct function

Non-copyable Classes

- A class is made non-copyable by declaring a private copy constructor and a private copy assignment operator and qualifying them with the delete keyword

```
class ISensor {
public:
    // A virtual destructor is crucial for any class intended for
polymorphic use.
    // It ensures that the correct derived class destructor is
called when an
    // object is deleted through a base class pointer.
    virtual ~ISensor() = default;

    // --- Rule of Five: Non-copyable and Non-movable ---
    // By deleting the copy and move operations, we prevent object
slicing and
    // make the interface's ownership semantics clear. It should
only be
    // managed via pointers (like std::unique_ptr).
    ISensor(const ISensor&) = delete;
    ISensor& operator=(const ISensor&) = delete;
    ISensor(ISensor&&) = delete;
    ISensor& operator=(ISensor&&) = delete;

    /**
     * @brief Reads a value from the sensor.
     * @return The sensor reading as a float.
     */
    virtual float read() const = 0; // Pure virtual function

    /**
     * @brief Gets the name of the sensor.
     * @return The sensor's name as a string.
     */
    virtual const char* get_name() const = 0; // Pure virtual
function

protected:
    // Protected constructor to prevent direct instantiation but
allow derivation.
    ISensor() = default;
};
```

Constant Functions

- Functions that do not modify any class-internal data and can be declared as const
- A client using a constant instance of class can only call constant members of the class
- A constant member function can modify a non-constant variable if that variable is qualified with the mutable keyword

```
float read() const override {
        // In a real system, this would involve
I2C/SPI/1-Wire communication.
        std::cout << "  (Reading from Temperature
Sensor hardware...)\n";
        return 24.5f; // degrees Celsius
    }

    const char* get_name() const override {
        return "TemperatureSensor";
    }
```

Friend classes & Friend Functions

- Friend Function: A global function that is granted access to a class's private members.
- Friend Class: An entire class that is granted access to another class's private members.

```
class TemperatureSensor : public ISensor {
public:
    // Grant SensorMonitor access to our private members.
    friend class SensorMonitor;
.
.
.

class HumiditySensor : public ISensor {
public:
    // Grant the global function
`reset_sensor_calibration` access to our private members.
    friend void reset_sensor_calibration(HumiditySensor&
sensor);
.
.
.
```

Template Functions

- Templates can be used in code to mainly re-use the code for different types of data
- The compiler instantiates it for a known type by filling in the template code corresponding to its template parameters at compiler time

```
template<typename T>
T add(const T& a, const T& b)
{
    return a + b;
}

// Class Template Argument Deduction (CTAD) since C++17
const int n = add(1, 2); // 3
const auto s = add(std::string("abc"),
std::string("xyz")); // "abcxyz".
```

Template Class Type

- Template classes can be very convenient for making re-usable and scalable objects

```
/**
 * @brief A simple, templated class representing a 2D Cartesian coordinate.
 *
 * This class can use any arithmetic type for its coordinates, making it
 * flexible for various use cases (e.g., int for grid-based logic, double
 * for precise geometric calculations).
 */
template<typename T>
class Point {
public:
    // Ensure that this class is only instantiated with arithmetic types.
    static_assert(std::is_arithmetic_v<T>, "Point only supports arithmetic
types.");

    /**
     * @brief Default constructor. Initializes the point to the origin
(0.0, 0.0).
     */
    constexpr Point() : x(T{}), y(T{}) {}

    /**
     * @brief Constructs a Point with specified coordinates.
     * @param x_val The initial x-coordinate.
     * @param y_val The initial y-coordinate.
     */
    constexpr Point(T x_val, T y_val) : x(x_val), y(y_val) {}

    T x; ///< The x-coordinate of the point.
    T y; ///< The y-coordinate of the point.

    /**
     * @brief Calculates the Euclidean distance from the origin (0,0) to
this point.
     * @note This function is only enabled for floating-point Point types
to
     *       avoid incorrect results from integer truncation.
     * @return The distance as a double.
     */
    template<typename U = T,
             typename = std::enable_if_t<std::is_floating_point_v<U>>>
    double distance_from_origin() const {
        return std::sqrt(x * x + y * y);
    }
};
```

CTAD

- A major C++17 feature that dramatically simplifies the way we create objects from class templates.
- Before C++17, a template function call or class template object creation would need template arguments specified in angle brackets <>

```
// C++17 Deduction Guide:
// When constructing a Point with two different arithmetic types (e.g., int and double),
// deduce the Point's type to be the "common type" of the two.
// For Point(5, 3.14), the common type is double, so it deduces Point<double>.

template<typename T1, typename T2>
Point(T1, T2) -> Point<std::common_type_t<T1, T2>>;



// CTAD deduces Point<int>
Point p1(3, 4);

// CTAD deduces Point<double>
Point p2{5.0, 12.0};

// CTAD deduces Point<double> based on the deduction guide
Point p1(3, 4.5);
```

Static Polymorphism

- Static polymorphism (compile time) is facilitated by templates
- The compiler resolves calls during compilation. This means it has no runtime overhead (no vtable lookups) and allows for better optimization, like inlining.

```
// --- 1. The Concrete Implementations (No Base Class) ---

/**
 * @brief A concrete implementation for a temperature sensor.
 * Note that it does not inherit from any base class.
 */
class TemperatureSensor {
public:
    float read() const {
        // In a real system, this would involve I2C/SPI/1-Wire communication.
        std::cout << "  (Reading from Temperature Sensor hardware...)\n";
        return 24.5f; // degrees Celsius
    }

    const char* get_name() const {
        return "TemperatureSensor";
    }
};

/**
 * @brief A concrete implementation for a humidity sensor.
 * It also does not inherit from any base class.
 */
class HumiditySensor {
public:
    float read() const {
        // In a real system, this would be different hardware communication.
        std::cout << "  (Reading from Humidity Sensor hardware...)\n";
        return 55.2f; // percent relative humidity
    }

    const char* get_name() const {
        return "HumiditySensor";
    }
};

// --- 2. The Generic "Application Logic" via a Template ---

template <typename SensorType>
void process_sensor(const SensorType& sensor) {
    float value = sensor.read();
    std::cout << "Device '" << sensor.get_name() << "' reports value: " << value
<< "\n\n";
}

// Create concrete objects on the stack. No pointers or dynamic allocation needed.
    TemperatureSensor temp_sensor;
    HumiditySensor humidity_sensor;

    // Call the generic function with different concrete types.
    // The compiler resolves these calls at compile time.
    process_sensor(temp_sensor);
    process_sensor(humidity_sensor);
```

Variadic Templates

- C++11 feature that allows you to write functions and classes that take a variable number of arguments in a type-safe way
- The classic way to process variadic arguments is with recursion. However, C++17 introduced fold expressions, which provide a much more concise and often more efficient way to work with them.

```
// --- 1. The Classic Recursive Approach ---

// Base case: This function is called when the parameter pack `args` is empty,
// which terminates the recursion.
void print_recursive() {
    std::cout << std::endl; // End with a newline
}

/**
 * @brief A recursive variadic template function that prints all its arguments.
 *
 * This function "peels off" the first argument, prints it, and then recursively
 * calls itself with the rest of the arguments.
 *
 * @tparam T The type of the first argument.
 * @tparam Args A parameter pack representing the types of the remaining arguments.
 * @param first The first argument to print.
 * @param args The pack of remaining arguments.
 */
template <typename T, typename... Args>
void print_recursive(T first, Args... args) {
    std::cout << first << " "; // Print the first argument
    print_recursive(args...);  // Recurse with the rest of the arguments
}
// --- 2. The Modern C++17 Fold Expression Approach ---

/**
 * @brief A variadic template function using a C++17 fold expression.
 *
 * This is much more concise and often more efficient than the recursive method.
 * The fold expression `(std::cout << ... << args)` expands at compile time
 * into a series of `<<` operations.
 */
template <typename... Args>
void print_modern(const Args&... args) {
    // The fold expression expands to: (std::cout << arg1 << " " << arg2 << " " << ...)
    ((std::cout << args << " "), ...);
    std::cout << std::endl;
}
```

Template Meta-programming Basics

- Templates allow computations at compile-time.
- Metaprogramming enables type traits and optimizations.

```
template<int N>
struct factorial {
  static const int value = N * factorial<N-1>::value;
};

// Terminating sequence
template<>
struct factorial<0> {
  static const int value = 1;
};

constexpr int val = factorial<5>::value; // 120
```

```
# armv7 assembly
main:
       push {r11, lr}
       mov r11, sp
       sub sp, sp, #8
       ldr r0, .LCPI0_0
       str r0, [sp, #4]
       ldr r0, .LCPI0_1
       str r0, [sp]
       ldr r0, .LCPI0_0
       mov sp, r11
       pop {r11, pc}
.LCPI0_0:
       .long 0
.LCPI0_1:
       .long 120
```

Template Meta-programming … *cont.*

- Factorial without template metaprogramming

```
unsigned long long
factorial_iterative(int n) {
	unsigned long long result = 1;
	for (int i = 2; i <= n; ++i) {
		result *= i;
	}
	return result;
}
```

```
factorial_iterative(int):
	push {r11, lr}
	mov r11, sp
	sub sp, sp, #24
	str r0, [r11, #-4]
	mov r0, #0
	str r0, [sp, #12]
	mov r0, #1
	str r0, [sp, #8]
	ldr r0, .LCPI0_0
	str r0, [sp, #4]
.LBB0_1:
	ldr r0, [sp, #4]
	ldr r1, [r11, #-4]
	cmp r0, r1
	bgt .LBB0_4
	ldr r2, [sp, #4]
	asr r12, r2, #31
	ldr r3, [sp, #8]
	ldr r0, [sp, #12]
	umull r1, lr, r3, r2
	mla r3, r3, r12, lr
	mla r0, r0, r2, r3
	str r1, [sp, #8]
	str r0, [sp, #12]
	ldr r0, [sp, #4]
	add r0, r0, #1
	str r0, [sp, #4]
	b .LBB0_1
.LBB0_4:
	ldr r0, [sp, #8]
	ldr r1, [sp, #12]
	mov sp, r11
	pop {r11, pc}
.LCPI0_0:
	.long 2
```

Type Traits

- Type traits allow decisions based on types at compile-time.
- They are widely used in STL and generic libraries.

```
template <typename uShortTy, typename uLargeTy>
constexpr uLargeTy makeLarge(const uShortTy us, const uShortTy ul)
{
    // Ensure that the template is only instantiated with unsigned
integer types.
    static_assert(std::is_unsigned<uShortTy>::value, "uShortTy must
be an unsigned type.");
    static_assert(std::is_unsigned<uLargeTy>::value, "uLargeTy must
be an unsigned type.");

    constexpr int uShortDigits =
std::numeric_limits<uShortTy>::digits;
    constexpr int uLargeDigits =
std::numeric_limits<uLargeTy>::digits;

    // Ensure the large type is exactly twice the size of the short
type.
    static_assert(uLargeDigits == uShortDigits * 2, "uLargeTy must
have twice the bits of uShortTy.");

    const uLargeTy ulShifted = static_cast<uLargeTy>(ul) <<
uShortDigits;

    return static_cast<uLargeTy>(us) | ulShifted;
}
```

Variable Templates

- Consider the mathematical computation for Stirling's approximation of Gamma function for large arguments ($\Gamma(z) \approx (z-1)!$ for large z)

$$\Gamma(z) \approx \sqrt{2\pi} * z^{(z-0.5)} * e^{-z} * e^{\frac{1}{12z} - \frac{1}{360z^3}}$$

- This template approximation of the Gamma function can be instantiated for float, double and long double.

```
// Can also use C++20's standardized constants from the <numbers>
library
template<typename T>
constexpr T pi =
T(3.1415926535'8979323846'2643383279'5028841972L);

template<typename T>
constexpr T e =
T(2.7182818284'5904523536'0287471352'6624977572L);

template<typename T>
T stirling_gamma_approx_order2(T z) {
    // Pre-calculate terms for clarity and efficiency
    const T one_over_z = 1.0 / z;
    const T one_over_z_cubed = one_over_z * one_over_z *
one_over_z;

    // The series expansion part: exp(1/(12z) - 1/(360z³))
    const T series_expansion = std::pow(e<T>, ((1.0 / 12.0) *
one_over_z - (1.0 / 360.0) * one_over_z_cubed));

    // The main Stirling formula part: sqrt(2π) * z^(z-0.5) *
e^(-z)
    const T main_term = std::sqrt(2.0 * pi<T>) * std::pow(z, z -
0.5) * std::pow(e<T>, (-z));

    return main_term * series_expansion;
}
```

Variable Templates ... *cont.*

- Choosing different values based on type_traits

```
// A more advanced variable template to provide a "safe" default
value for a type.
// This uses `if constexpr` (C++17) to select the value at
compile time.
// The value is determined by an immediately-invoked lambda.
template<typename T>
inline constexpr T safe_default_v = [] {
    if constexpr (std::is_arithmetic_v<T>) {
        return T{0}; // 0 for int, 0.0 for double, etc.
    } else if constexpr (std::is_pointer_v<T>) {
        return nullptr; // nullptr for any pointer type
    } else {
        // For other types like std::string, it will be default-
constructed.
        // This branch is only compiled if the type T is not
arithmetic or a pointer.
        return T{};
    }
}(); // Note the () - this invokes the lambda immediately.

// Helper function to print values
template<typename T>
void print_defaults() {
    // Here we instantiate the variable template for type T
    T val = safe_default_v<T>;
    std::cout << "Default for type '" << typeid(T).name() << "'
is: " << val << std::endl;
}
```

```
print_defaults<int>();           // Default for type 'i' is: 0
print_defaults<double>();        // Default for type 'd' is: 0
print_defaults<int*>();          // Default for type 'Pi' is: 0x0
print_defaults<std::string>();   // Default for type
'NSt3__112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEEE'
is: ""
```

SFINAE and enable_if

- enable_if helps control template instantiation.
- Used to enable/disable functions depending on conditions.

```
#include <type_traits>

// Overload 1: Enabled only for integral types (int, char,
long, etc.)
// The return type is `void`, but only if
`std::is_integral<T>::value` is true.

template <typename T>
typename std::enable_if<std::is_integral<T>::value,
void>::type
log_value(T val) { /* ... */ }
```

Concepts – A better way than enable_if

- Concepts are a core language feature designed specifically for expressing constraints on templates
- Concepts tend to be more readable, expressive, and user-friendly

```
#include <type_traits>

// Define the concept
template <typename T>
concept Integral = std::is_integral_v<T>;

// Use it directly in the function signature (terse syntax)
void log_value(Integral auto val) { /* ... */ }
```

OPTIMIZING C++ FOR EMBEDDED SYSTEMS

Concepts, Examples, and Applications

Compiler Optimizations

- Compiler optimization allows for flexible tuning of compiler code generation
- Compilation can be optimized for space, speed, or both

Optimization	Code Size (bytes)	Runtime (µs)
Space (with -Oz)	3884	0.014984
Speed (with -O2)	5009	0.011738
Speed (with -O3)	5041	0.010964

Table comparing optimizations, code size and runtime for a CRC232-MPEG2 algorithm

Using Assembly Listings

- Assembly listings allow us to break down the high-level program to its machine level opcodes.
- A basic understanding of assembly listings makes it possible to analyze the time critical

```
template<int N>
struct factorial {
  static const int value = N * factorial<N-1>::value;
};

// Terminating sequence
template<>
struct factorial<0> {
  static const int value = 1;
};

constexpr int val = factorial<5>::value; // 120
```

```
# armv7 assembly generated using command
# objdump –j .text –S <filename>.o > <filename>.txt

main:
       push {r11, lr}
       mov r11, sp
       sub sp, sp, #8
       ldr r0, .LCPI0_0
       str r0, [sp, #4]
       ldr r0, .LCPI0_1
       str r0, [sp]
       ldr r0, .LCPI0_0
       mov sp, r11
       pop {r11, pc}
.LCPI0_0:
       .long 0
.LCPI0_1:
       .long 120
```

Using Map files

- Detailed information about the addresses and symbol sizes
- Along with assembly listings, map files can be useful for iteratively finding best trade-off between space and speed

```
# VTables for Dynamic polymorphism example
# Symbols:
# Address    Size         File  Name
...
0x100008178  0x00000030   [  1] __ZTV17TemperatureSensor
...
0x1000081D0  0x00000030   [  1] __ZTV7ISensor
...
0x100008200  0x00000030   [  1] __ZTV14HumiditySensor
...
```

```
# Symbols:
# Address    Size         File  Name
...
0x1000041C8  0x00000014   [  1] __ZTS17TemperatureSensor
...
0x100008178  0x00000030   [  1] __ZTV17TemperatureSensor
...
0x1000081B8  0x00000018   [  1] __ZTI17TemperatureSensor
...
```

Using Map files … *cont.*

- Map files are the ultimate truth for analyzing the size and offsets of symbols in your code
- This helps greatly in making design decisions based on complexity and resources

Feature	polymorphism_embedded_example.map (Dynamic)	static_polymorphism_embedded_example.map (Static)
Polymorphism Mechanism	virtual functions, base class pointers	C++ templates
Code Size (.text)	**14,140 bytes** (0x373C)	**6,440 bytes** (0x1928)
V-Tables (__ZTV)	**Present** (3 tables)	**Absent**
RTTI (__ZTI)	**Present** (3 objects)	**Absent**
Heap Usage	**Yes** (via std::unique_ptr, std::vector)	**No** (stack allocation only)
Complexity	High (many STL symbols for memory management)	Low (direct function calls)

```
# Comparison of the map files generated for the Dynamic polymorphism
and Static polymorphism example
# Generated by the command: clang++ -std=c++26 <filename>.cpp -o
<filename>.o -Wl,-map, <filename>.map
```

Name mangling & De-mangling

- Same name symbols can exist in different namespaces or in different classes
- In order to create non-conflicting symbols, the C++ compiler needs to decorate the symbol names
- These decorated internal symbol names can be very long, and so the process has come to be known as name mangling.
- c++filt tool can be used for demangling the symbol names

```
# Demangling the symbol
__ZNK17TemperatureSensor4readEv

$ c++filt -_ __ZNK17TemperatureSensor4readEv
TemperatureSensor::read() const
```

Utilizing ROM efficiently

- In the absence of a loader to manage the memory, the linker needs to be told where to where to place the symbols using a linker script (for GCC/Clang) or a scatter-gather file (for ARM/Keil)
- All the text section and the rodata would be placed in ROM memory for efficient utilization of the limited RAM resource

```
/* --- 1. Define the Entry Point of the Application --- */
/* This is the first function that gets executed after a reset. */
ENTRY(Reset_Handler)

/* 2. Define the memory layout of the microcontroller */
MEMORY
{
  FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 128K  /* Read-only, Executable */
  SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 32K.  /* SRAM (RAM): Read-Write */
}

/* 3. Define the output sections and tell the linker where to put them */
SECTIONS
{
  /* The .text section (program code) goes into FLASH */
  .text :
  {
    *(.text)          /* All .text sections from all input object files */
    *(.text*)         /* e.g., .text.main */
    . = ALIGN(4);
  } > FLASH

  /* The .rodata section (read-only data) also goes into FLASH */
  .rodata :
  {
    *(.rodata)        /* All .rodata sections (const variables, string
literals) */
    *(.rodata*)
    . = ALIGN(4);
  } > FLASH

/* The .data section (initialized global/static variables) is more complex.
     The initial values are stored in FLASH but copied to RAM at startup. */
  .data :
  {
    *(.data)
    *(.data*)
    . = ALIGN(4);
  } > SRAM AT> FLASH /* Place in RAM, but load the initial values from FLASH */

/* The LMA of .data is stored immediately after .text in FLASH */
    _sidata = LOADADDR(.data);

/* The .bss section (uninitialized global/static variables) just needs space in
RAM */
  .bss :
  {
    *(.bss)
    *(.bss*)
    . = ALIGN(4);
  } > SRAM
}
```

Custom Memory Management Using Placement-new

- Placement-new allows the programmer to explicitly control a dynamically created object's physical address
- Unlike the regular new operator, which both allocates memory and then constructs an object, placement new only performs the construction step.

```
class MyObject {
public:
    MyObject(int id, const std::string& name) : id_(id),
name_(name) {
        std::cout << "MyObject constructor called for ID " <<
id_ << " at address " << this << std::endl;
    }

    ~MyObject() {
        std::cout << "MyObject destructor called for ID " << id_
<< " at address " << this << std::endl;
    }

    void print() const {
        std::cout << "  - ID: " << id_ << ", Name: '" << name_
<< "'\n";
    }


// ------ Application -------
// 1. Pre-allocate a raw memory buffer.
    // This buffer is large enough and correctly aligned to hold
two MyObject instances.
    // Using std::byte is the modern C++ way to represent raw
memory.
    alignas(MyObject) std::byte memory_buffer[2 *
sizeof(MyObject)];
    std::cout << "Pre-allocated a memory buffer at address: " <<
static_cast<void*>(memory_buffer) << std::endl << std::endl;

    // 2. Use placement new to construct objects in the buffer.
    // This calls the constructor but does NOT allocate memory.
    MyObject* obj1 = new (memory_buffer) MyObject(101, "First");
    MyObject* obj2 = new (memory_buffer + sizeof(MyObject))
MyObject(202, "Second");
```

Lambda Expressions

- Lambda expressions offer locality of code, in that the logic of the function is inlined at the place of call
- Provide Stateful Callbacks with its power of capturing variables from surrounding scope

```
class GPIO_Manager {
public:
    /**
     * @brief Registers a callback function to be executed when a pin
changes state.
     * @param pin The pin number to monitor.
     * @param callback The function or lambda to execute on an
interrupt.
     */
    void register_interrupt_handler(int pin, InterruptCallback
callback) {
        std::cout << "[GPIO] Registered handler for pin " << pin <<
".\n";
        interrupt_handlers_[pin] = std::move(callback);
    }

private:
    std::map<int, InterruptCallback> interrupt_handlers_;
};


GPIO_Manager gpio;
    int button_press_count = 0;
    bool system_armed = true;

    // --- The Lambda Expression ---
    // Here, we define the behavior for a button press directly where
we register it.
    // The lambda captures `button_press_count` and `system_armed` by
reference,
    // allowing it to interact with the state of `main`.
    gpio.register_interrupt_handler(12,
        // Capture by reference to modify variables in main's scope.
        [&button_press_count, &system_armed]() {
            std::cout << "[CALLBACK] Executing handler for pin 12.\n";
            if (system_armed) {
                button_press_count++;
                std::cout << "[CALLBACK] System is armed. Incrementing
count.\n";
            } else {
                std::cout << "[CALLBACK] System is disarmed. Ignoring
press.\n";
            }
        });
```

Register Access

- Register addresses can be conveniently defined with compile-time constant static integral members of a class type or, using the constexpr keyword
- Register access can be made versatile by using templates for making 32 bit and 64bit register accesses.

```
// --- 1. Define Memory-Mapped Addresses from the Datasheet ---
// Using `constexpr` ensures they are compile-time constants.
namespace peripheral_addresses {
    constexpr std::uintptr_t GPIOA_BASE = 0x40020000;
    constexpr std::uintptr_t GPIOB_BASE = 0x40020400;
    // ... other peripherals
}
```

Register Access ... *cont.*

```
/**
 * @brief A handler class for performing safe, masked register operations.
 *
 * This class encapsulates the logic for read-modify-write operations,
 * which is a very common pattern in embedded programming.
 * @tparam AddressType The integer type used for memory addresses (e.g.,
uintptr_t).
 */
template<typename AddressType>
class RegisterAccessHandler {
public:
    /**
     * @brief Constructs the handler with the base address of a peripheral.
     * @param base_address The starting memory address of the peripheral.
     */
    explicit RegisterAccessHandler(AddressType base_address) :
base_address_(base_address) {}

    /**
     * @brief Performs a masked write to a register.
     *
     * This is a "read-modify-write" operation. It reads the current
register value,
     * clears the bits specified by the mask, sets the bits from the input
value,
     * and writes the result back.
     *
     * @tparam T The type of the register (e.g., uint32_t).
     * @param inOffset The byte offset of the register from the peripheral's
base address.
     * @param inMask The bitmask indicating which bits to modify.
     * @param inValue The value to write to the masked bits.
     */
    template<typename T>
    void maskedRegisterWrite(AddressType inOffset, T inMask, T inValue)
const;

    /**
     * @brief Performs a masked read from a register.
     *
     * @tparam T The type of the register (e.g., uint32_t).
     * @param inOffset The byte offset of the register from the peripheral's
base address.
     * @param inMask The bitmask indicating which bits to read.
     * @param outValue A reference to store the masked result.
     */
    template<typename T>
    void maskedRegisterRead(AddressType inOffset, T inMask, T& outValue)
const;
```

Start-up Code using C++

- The uninitialized and initialized globals and static variables need to be zero-filled and initialized respectively before main() is called
- The static constructor instances need to be initialized – compiler generated function stored in special linker sections named such as ctors or init_arrays

```
// Zero-Filling uninitialized data
// Linker-defined begin and end of the .bss section.
extern std::uint8_t* _bss_begin;
extern std::uint8_t* _bss_end;
void init_bss()
{
    // Clear the bss segment.
    std::fill(_bss_begin, _bss_end, 0U);
}
```

```
// Initializing initialized data from ROM
// Linker-defined begin of rodata.
extern std::uint8_t* _rodata_begin;

// Linker-defined begin and end of data.
extern std::uint8_t* _data_begin;
extern std::uint8_t* _data_end;

void init_data()
{
    // Calculate the size of the data section.
    const std::size_t count = _data_end - _data_begin;
    // Copy the rodata section to the data section.
    std::copy(_rodata_begin,
        _rodata_begin + count, _data_begin);
}
```

```
// Initializing static constructor instances
using function_type = void(*)();

// Linker-defined begin and end of the ctors.
extern function_type _ctors_begin[];
extern function_type _ctors_end [];

void init_ctors()
{
    std::for_each(_ctors_begin, _ctors_end,
                  [](const function_type& pf)
                  {
                      pf();
                  });
}
```

Static Initialization Considerations

- All of the non-subroutine-local statics must be initialized before the call to the main()
- The non-subroutine-local statics are guaranteed to be initialized before any function in the containing file uses it.
- No rule to govern the order of initialization of different files, so the access of a static instance initialized in a separate file could lead to crashes if the instance doesn't get initialized before-hand

```
class Logger {
public:
    /**
     * @brief Provides the single, global point of access to the
Logger instance.
     *
     * C++11 guarantees that the local static `instance` is
initialized only once,
     * on the first call to this function, and that its
initialization is thread-safe.
     * This is the core of the pattern that solves the static
initialization order fiasco.
     * @return A reference to the single Logger instance.
     */
    static Logger& getInstance() {
        static Logger instance;
        return instance;
    }

    void log(const std::string& message);

    // Delete copy/move operations to enforce the "one instance"
rule.
    Logger(const Logger&) = delete;
    Logger& operator=(const Logger&) =delete;
    Logger(Logger&&) = delete;
    Logger& operator=(Logger&&) = delete;

private:
    Logger();  // Private constructor
    ~Logger(); // Private destructor
    std::mutex log_mutex_; // To make the log() method itself
thread-safe
};
```

Multithreading in C++

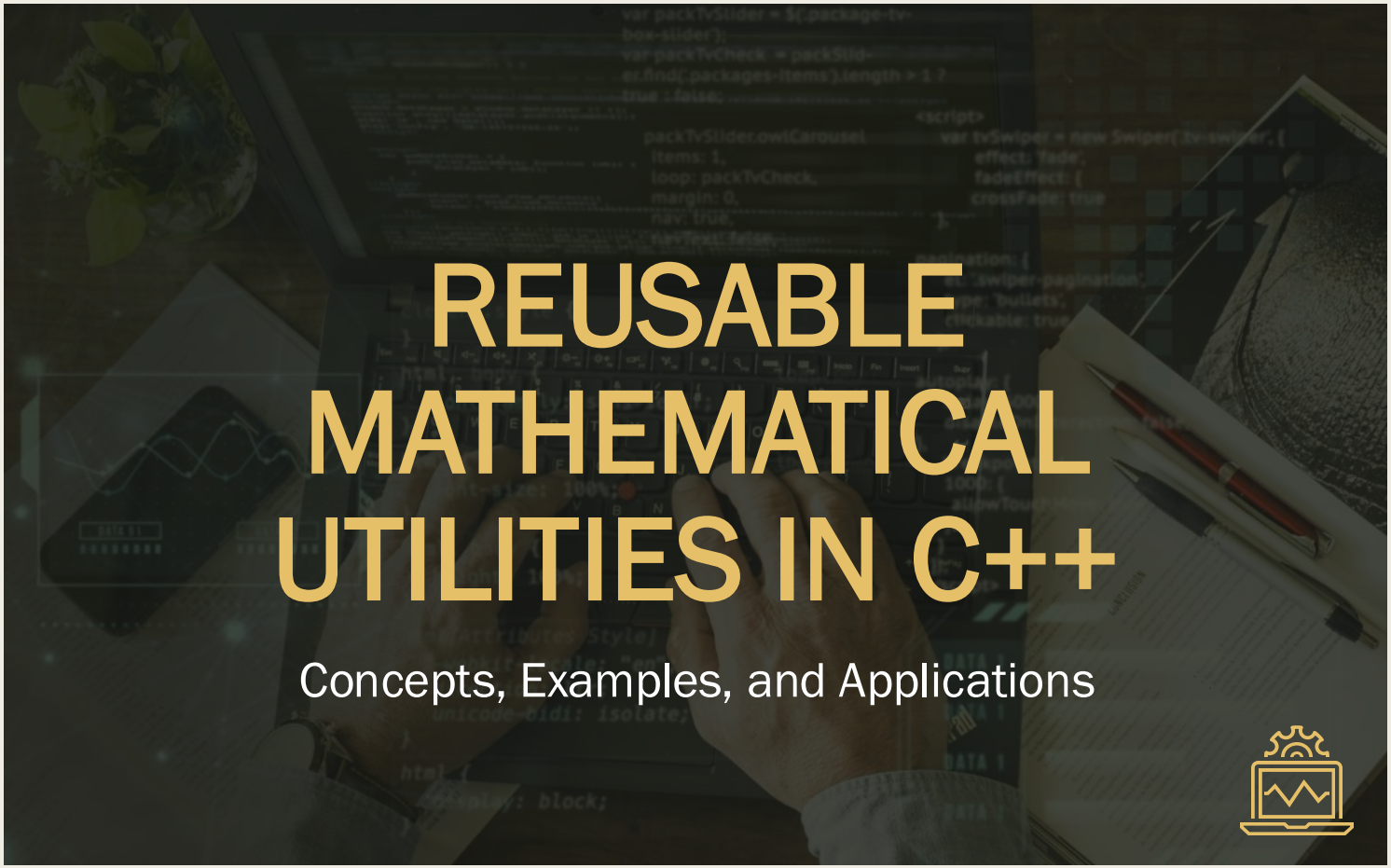
- C++ supports multithreading through its thread support library
- Full support for the C++ thread library (<thread>, <mutex>, etc.) on microcontroller compilers is not common and is almost entirely dependent on the presence of an underlying Real-Time Operating System (RTOS).
- Use of C++ Wrappers Around Native RTOS APIs

```
class IRunnable {
public:
    virtual ~IRunnable() = default;
    virtual void run() = 0;
};

// This is the C-style entry point that FreeRTOS will call.
// It is declared `extern "C"` to prevent C++ name mangling.
extern "C" void task_entry_point_trampoline(void* task_instance) {
    // Cast the void pointer to our abstract base class and call the
virtual run() method.
    static_cast<IRunnable*>(task_instance)->run();
}

/**
 * @brief A generic C++ wrapper for a FreeRTOS task.
 *
 * This class handles task creation and uses the CRTP (Curiously Recurring
 * Template Pattern) to call the `run()` method of the derived class.
 * This avoids breaking type safety with void pointers.
 */
template <class T>
class FreeRTOSTask : public IRunnable {
public:
    // Prevent copying
    FreeRTOSTask(const FreeRTOSTask&) = delete;
    FreeRTOSTask& operator=(const FreeRTOSTask&) = delete;

    // Start the FreeRTOS task
    void start(const char* name, int priority = 1, int stack_size = 256) {
        xTaskCreate(
            task_entry_point_trampoline, // Pointer to the C-style
trampoline function
            name,              // A name for the task
            stack_size,        // Stack depth
            this,              // Pass a pointer to this object (as an
IRunnable*)
            priority,          // Task priority
            &task_handle_      // Pointer to the task handle
        );
    }
.
.
```

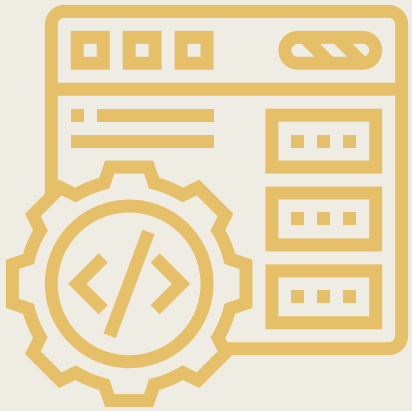


REUSABLE MATHEMATICAL UTILITIES IN C++

Concepts, Examples, and Applications

Floating-Point Math

- Most C++ compilers provide conform with IEEE-754:2008 standard
- However, many microcontrollers don't provide a floating-point unit (FPU)
- FPUs aid in making floating-point arithmetic as efficient as integer calculations
- In absence of FPUs, software floating-point emulation libraries can be used, however, it may introduce a large amount of library code in the executable and make the application slow
- It would even be resource-intensive



Mathematical Constants

- C++20 provides a rich set of common mathematical constants √2, π, log 2, e, etc. in the <numbers> header
- Standard mathematical constants can also be implemented using variable templates which can be instantiated as float, double, or long double.

```
// Archimedes' constant (π ≈ 3.14159... )
template<typename T>
constexpr T pi =
T(3.1415926535'8979323846'2643383279'5028841972L);

// Natural logarithmic base (e ≈ 2.71828... )
template<typename T>
constexpr T e =
T(2.7182818284'5904523536'0287471352'6624977572L);
```

Complex values

- The C++ standard library supports complex-valued mathematics with its template data type std::complex.
- The std::complex data type is defined in <complex> and specified for (and only for) the built-in types float, double and long double.

```
// 1. Creating complex numbers
// std::complex is a template class, usually used with
float, double, or long double.
using Complex = std::complex<double>;

Complex z1(3.0, 4.0); // Represents 3 + 4i
Complex z2(5.0, -2.0); // Represents 5 - 2i
Complex i(0.0, 1.0);  // The imaginary unit ‘i’

// 2. Basic Arithmetic
// The standard arithmetic operators (+, -, *, /) are
overloaded.
std::cout << "--- Basic Arithmetic ---" << std::endl;
Complex sum = z1 + z2; // 8+2i
Complex product = z1 * z2; // (3+4i)*(5-2i) = 15-6i+20i-8i^2
= 23+14i

// 4. Creating from Polar Coordinates
// Euler's formula: e^(i*theta) = cos(theta) + i*sin(theta)
// A complex number can be represented as r * e^(i*theta)
std::cout << "--- Polar Coordinates ---" << std::endl;
double magnitude = 2.0;
double angle = pi / 2.0; // 90 degrees

// std::polar creates a complex number from a magnitude and
an angle (in radians).
Complex z_polar = std::polar(magnitude, angle); //
(0.0000,2.0000) -> (2i)
```

Fixed-Point Math

- Many microcontroller platforms don’t provide a native FPU HW to perform floating-point calculations
- In order to avoid the performance penalty of using the floating-point SW libraries instead, fixed-point calculations can be used
- A fixed-point number is an integer-based data type representing a fractional number, optionally signed, having a fixed number of integer digits to the left of the decimal point and another fixed number of fractional digits to the right of the decimal point

```
/**
 * @brief A class for fixed-point numbers.
 *
 * @tparam IntegerType The underlying integer type (e.g., int32_t).
 * @tparam FractionalBits The number of bits to use for the fractional
part.
 */
template<typename IntegerType, int FractionalBits>
class FixedPoint {
public:
    // The scaling factor is 2^FractionalBits.
    static constexpr IntegerType Scale = 1LL << FractionalBits;

    // Default constructor
    FixedPoint() : raw_value_(0) {}

    // Constructor from a floating-point number
    FixedPoint(double val) : raw_value_(static_cast<IntegerType>(val *
Scale)) {}

    // Conversion operator to get the value as a double
    explicit operator double() const {
        return static_cast<double>(raw_value_) / Scale;
    }

    // --- Overloaded Arithmetic Operators ---

    // Addition: The underlying integers can be added directly.
    FixedPoint operator+(const FixedPoint& other) const {
        return from_raw(raw_value_ + other.raw_value_);
    }

    // Subtraction: The underlying integers can be subtracted directly.
    FixedPoint operator-(const FixedPoint& other) const {
        return from_raw(raw_value_ - other.raw_value_);
    }
```

Fixed-Point Math ... *cont.*

```
    // Multiplication: (a * 2^N) * (b * 2^N) = (a*b) * 2^(2N).
    // We must shift the result right by N to re-normalize.
    // We use a wider integer type to prevent overflow during multiplication.
    FixedPoint operator*(const FixedPoint& other) const {
        using WiderType = __int128_t; // Use a wider type for intermediate
calculation
        WiderType temp = static_cast<WiderType>(raw_value_) *
other.raw_value_;
        return from_raw(static_cast<IntegerType>(temp >> FractionalBits));
    }

    // Division: (a * 2^N) / (b * 2^N) = a/b.
    // To preserve precision, we pre-shift the numerator left by N.
    // We use a wider integer type to prevent overflow during the pre-shift.
    FixedPoint operator/(const FixedPoint& other) const {
        using WiderType = __int128_t;
        WiderType temp = (static_cast<WiderType>(raw_value_)
                          << FractionalBits) / other.raw_value_;
        return from_raw(static_cast<IntegerType>(temp));
    }

private:
    // Private constructor to create a FixedPoint from a raw scaled value.
    // This is used internally by the operators.
    explicit FixedPoint(IntegerType raw) : raw_value_(raw) {}

    // Static factory function for the private constructor.
    static FixedPoint from_raw(IntegerType raw) {
        return FixedPoint(raw);
    }

    IntegerType raw_value_;
};
// We'll use a 64-bit integer with 32 bits for the fractional part (Q31.32).
// This gives us 1 sign bit, 31 integer bits, and 32 fractional bits.
// Precision is ~9.6 decimal digits (32 * log10(2)).
using Fp = FixedPoint<int64_t, 32>;

// --- Create fixed-point numbers from doubles ---
Fp a = 3.5;
Fp b = -2.25;

// --- Perform arithmetic ---
Fp sum = a + b; // 1.2500 | 1.2500 (float verification)
Fp diff = a - b; // 5.7500 | 5.7500 (float verification)
Fp prod = a * b; // -7.8750 | -7.8750 (float verification)
Fp quot = a / b; // -1.5556 | -1.5556 (float verification)
```


QUESTIONS?