m3ta Documentation

Release

Christopher Crouzet

March 21, 2015

Contents

Welcome to the documentation of m3ta.

CHAPTER 1

Getting Started

1.1 Overview

m3ta is an header-only library regrouping utilities for template metaprogramming.

It relies on features implemented in C++11 and requires as such C++11 or above.

Check out the *Examples* section to see some use cases.

Note: The library has been tested only against Clang 3.5.0 but should work on any C++11 compliant compiler.

1.2 Examples

1.2.1 Function Call at Compile Time

Let's consider the case of 2D texture tiles to be used in the context of tile-based texture mapping within graphic renderers such as OpenGL. Such tiles must be square and it is often recommended for hardware optimisations and/or limitations to set their size to a power of two.

```
template<int T_size>
class TextureTile2D
{};
```

Defining the size as a template parameter force the user to provide it at compile time, allowing some possible further optimisations.

As it stands with this class definition, a 2D texture tile could be initialized by passing any value—power of two, or not.

```
TextureTile2D<256> powerOfTwoTexture;
TextureTile2D<123> nonPowerOfTwoTexture;
```

To avoid the risk of passing non power of two values by mistake, one could decide to use a power function such as the stardard `std::pow()`.

```
TextureTile2D<std::pow(2, 8)> powerOfTwoTexture;
```

Expect that this won't compile because the template parameter `T_size` needs to be set at compile time while the `std::pow()` function can only be ran at runtime. That's where the *m3ta::power()* function comes into play.

```
TextureTile2D<m3ta::power(2, 8)> powerOfTwoTexture;
```

This works because the *m3ta::power()* function—like all the other functions from this library—is defined with the `constexpr` qualifier introduced in C++11, allowing it to run either at runtime or at compile time, depending on the context. The context here being a template parameter, the function runs at compile time.

1.2.2 Tag Dispatching

Building on the previous example, let's assume that `TextureTile2D` has a method that does something where the computations involved could be highly optimized if the size passed as argument was a power of two less than 1024.

```
constexpr bool isPowerOfTwo(int value) { ... }

template<int T_size>
class TextureTile2D
{
public:
    void doSomething()
    {
        if (isPowerOfTwo(T_size) && T_size < 1024) {
            // Run optimized code.
        }
        else {
            // Run unoptimized code.
        }
    }
};
```

This runs fine but the conditional check is done here at runtime even though the value `T_size` is known at compile time and that the function `isPowerOfTwo` can also run at compile time. If the check was expensive, it could be worthwhile to compute it once for all at compile time.

```
constexpr bool isPowerOfTwo(int value) { ... }

template<int T_size>
class TextureTile2D
{
public:
    void
    doSomething()
    {
        doSomething_impl(m3ta::All<bool, isPowerOfTwo(T_size), T_size < 1024>());
    }

private:
    void
    doSomething_impl(std::true_type)
    {
        // Run optimized code.
    }

    void
    doSomething_impl(std::false_type)
    {
        // Run unoptimized code.
    }
};
```

Here the trait *m3ta::All* inherits from `std::true_type` (an alias of `std::integral_type<bool, true>`) if all the conditions passed to *m3ta::All* evaluate to *true*, and from `std::false_type` otherwise.

This is called the tag dispatching technique—it allows to pick either one of the implementations at compile time.

1.2.3 Overload Resolution Using SFINAE

```
template<typename T1, typename T2, typename = void>
struct AreComparableForEquality
{
    static constexpr bool value = false;
};

template<typename T1, typename T2>
struct AreComparableForEquality<
    T1,
    T2,
    typename std::enable_if<
        true,
        m3ta::PassT<void, decltype(std::declval<T1>() == std::declval<T2>())>
    >::type
>
{
    static constexpr bool value = true;
};
```

This `AreComparableForEquality` trait defined here allows to check if two types can be compared for equality using the operator ==. For example `AreComparableForEquality<int, float>::value` returns `true` while `AreComparableForEquality<std::string, float>::value` returns `false`.

This works by using the SFINAE technique over template parameters—if the code `std::declval<T1>() == std::declval<T2>()` is not a valid expression (no equality operator is defined between he types `T1` and `T2`), then the default overload holding the value `false` is picked, otherwise the second one is chosen at one condition: the specialization of the third parameter must return `void` as per the default template parameter from the first overload.

m3ta::PassT is a used here to conveniently test the SFINAE expression while always returning the required type.

1.2.4 Static Assert

Let's assume a `Divide` struct that would operate divisions between two integers at compile time. To disallow divisions by 0, a possibility could be to define a template specialization for the divisor value that would trigger an informative error message at compile time using `static_assert`.

```
template<int T_dividend, int T_divisor>
struct Divide
{
    static constexpr int value = T_dividend / T_divisor;
};

template<int T_dividend>
struct Divide<T_dividend, 0>
{
    static_assert(false, "Division by 0 not allowed.");
};
```

This code won't work because the `static_assert` will always be evaluated to *false* during the compilation—and hence will trigger a compilation error—even when no code path would lead to instantiate the version with that template specialization.

To get it to work, the evaluation of the `static_assert` needs to be slightly deferred by making it dependent on a type—any type really, including empty parameter packs—with the help of the *m3ta::dependenBool()* function.

```
template<int T_dividend, int T_divisor, typename ... T_Dummies>
struct Base
{
   static constexpr int value = T_dividend / T_divisor;
};

template<int T_dividend, typename ... T_Dummies>
struct Base<T_dividend, 0, T_Dummies ...>
{
    static_assert(
        m3ta::dependentBool<T_Dummies ...>(false),
        "Division by 0 not allowed."
    );
};

template<int T_dividend, int T_divisor>
struct Divide
    : public Base<T_dividend, T_divisor>
{};
```

1.2.5 The Nested Initializers Trick for Multidimensional Arrays

The built-in C/C++ arrays come with a convenient syntax to initialize them: the curly braces.

```
int array[2][3] = {
    0, 1, 2,
    3, 4, 5
};
```

The definition of `array` produces a multidimensional array that represents 2 arrays of 3 elements each.

The same array could have been written with an extra pair of brace for each inner dimension.

```
int array[2][3] = {
    {0, 1, 2},
    {3, 4, 5}
};
```

With the introduction of the initializer lists in C++11, this syntax is now usable within custom types. Reproducing the first syntax requires a constructor to accept a single `std::initializer_list` argument, while the nested braces syntax requires a parameter to be defined as nested initializer lists.

```
template<typename T, std::size_t ... T_dimensions>
class MultidimensionalArray
{
protected:
    using NestedInitializerLists =
        m3ta::NestedInitializerListsT<T, sizeof ... (T_dimensions)>;

public:
    static constexpr std::size_t
    size()
    {
        return m3ta::product(T_dimensions ...);
    }
```

```
    MultidimensionalArray(NestedInitializerLists lists)
    { ... }

private:
    std::array<T, size()> _data;
};
```

The *m3ta::NestedInitializerLists* traits allows to quickly define a new type with a specified number of `std::initializer_list` nested within each other, while the *m3ta::product()* function returns the total size of the multidimensional array.

From there, iterating through each element is not as simple as iterating over a linear container. Indeed, iterating through the `std::initializer_lists` at the top level with the function `begin()` returns pointers to the deeper levels. As such, the elements initialized with the nested braces syntax can only be iterated through a recursive approach.

```
template<typename T, std::size_t ... T_shape>
struct NestedInitializerListsProcessor;

template<typename T, std::size_t T_first, std::size_t ... T_others>
struct NestedInitializerListsProcessor<T, T_first, T_others ...>
{
    using NestedInitializerLists =
        m3ta::NestedInitializerListsT<T, 1 + sizeof ... (T_others)>;

    template<typename T_Function>
    static void
    process(NestedInitializerLists lists, T_Function function)
    {
        if (lists.size() > T_first) {
            throw std::invalid_argument(
                "Elements in excess within the initilizer list."
            );
        }

        for (auto nested : lists) {
            NestedInitializerListsProcessor<T, T_others ...>::
                process(nested, function);
        }

        if (T_first != lists.size()) {
            std::size_t count =
                m3ta::product(T_others ...) * (T_first - lists.size());

            for (; count > 0; --count) {
                function(static_cast<T>(0));
            }
        }
    }
};

template<typename T, std::size_t T_last>
struct NestedInitializerListsProcessor<T, T_last>
{
    using InitializerList = m3ta::NestedInitializerListsT<T, 1>;

    template<typename T_Function>
    static void
    process(InitializerList list, T_Function function)
```

```
    {
        if (list.size() > T_last) {
            throw std::invalid_argument(
                "Elements in excess within the initilizer list."
            );
        }

        std::for_each(list.begin(), list.end(), function);

        if (T_last != list.size()) {
            std::size_t count = T_last - list.size();
            for (; count > 0; --count) {
                function(static_cast<T>(0));
            }
        }
    }
};
```

The `NestedInitializerListsProcessor` helper can iterate through nested `std::initializer_lists` while allowing a custom function to be applied on each element.

With this in hands, it is now possible to fully implement the constructor for the nested braces syntax.

```
template<typename T, std::size_t ... T_dimensions>
class MultidimensionalArray
{
protected:
    using NestedInitializerLists =
        m3ta::NestedInitializerListsT<T, sizeof ... (T_dimensions)>;

public:
    static constexpr std::size_t
    size()
    {
        return m3ta::product(T_dimensions ...);
    }

    MultidimensionalArray(NestedInitializerLists lists)
    {
        auto iterator = _data.begin();
        NestedInitializerListsProcessor<T, T_dimensions ...>::
            process(
                lists,
                [&iterator](T value) { *(iterator++) = value; }
            );
    }

private:
    std::array<T, size()> _data;
};
```

CHAPTER 2

What's Available?

2.1 Reference

```
#include <m3ta/m3ta>
```

2.1.1 Constants

Regroups the utilities that are outputting constant values.

They are defined both as `constexpr` functions and as conventional traits inheriting from `std::integral_constant`.

All

```
#include <m3ta/all>
```

Checks if all the values evaluate to *true*.

See Also

Any and *None*.

Functions

m3ta::all

```
template<typename ... T_Values>
constexpr bool
all(T_Values ... values) noexcept
```

Template Parameters

- **T_Values** (automatically deduced) — Types of the values to check.

Function Parameters

- **values** – Variable number of values to check. Those can be of heterogeneous types.

Returns

Whether all the values evaluate to *true*.

Traits

m3ta::All

```
template<typename T, T ... T_values>
struct All
```

Template Parameters

- **T** – Type of the values to check.
- **T_values** – Variable number of values to check.

Member Types

type
The type `std::integral_constant<bool, value>` where *value* is the result of the function *m3ta::all()*.

value_type
The type `bool`.

Member Constants

static constexpr bool value
Whether all the values evaluate to *true*.

Aliases

m3ta::AllT

```
template<typename T, T ... T_values>
using AllT = typename All<T, T_values ...>::type;
```

Usage Examples

```
bool value1 = m3ta::all(true, true); // true
bool value2 = m3ta::all(false, false); // false
bool value3 = m3ta::all(true, false); // false
bool value4 = m3ta::all(true, 1, 2L, 4.0f, 9.0, 85.0L); // true

using Type1 = m3ta::AllT<bool, true, true>; // std::integral_constant<bool, true>
using Type2 = m3ta::AllT<bool, false, false>; // std::integral_constant<bool, false>
using Type3 = m3ta::AllT<bool, true, false>; // std::integral_constant<bool, false>
using Type4 = m3ta::AllT<bool, 1, 2, 4, 9, 85>; // std::integral_constant<bool, true>
```

Any

```
#include <m3ta/any>
```

Checks if any of the values evaluate to *true*.

See Also

All and *None*.

Functions

m3ta::any

```
template<typename ... T_Values>
constexpr bool
any(T_Values ... values) noexcept
```

Template Parameters

- **T_Values** (automatically deduced) — Types of the values to check.

Function Parameters

- **values** – Variable number of values to check. Those can be of heterogeneous types.

Returns

Whether any of the values evaluate to *true*.

Traits

m3ta::Any

```
template<typename T, T ... T_values>
struct Any
```

Template Parameters

- **T** – Type of the values to check.
- **T_values** – Variable number of values to check.

Member Types

type
The type `std::integral_constant<bool, value>` where *value* is the result of the function *m3ta::any()*.

value_type
The type `bool`.

Member Constants

static constexpr bool value
Whether any of the values evaluate to *true*.

Aliases

m3ta::AnyT

```
template<typename T, T ... T_values>
using AnyT = typename Any<T, T_values ...>::type;
```

Usage Examples

```
bool value1 = m3ta::any(true, true); // true
bool value2 = m3ta::any(false, false); // false
bool value3 = m3ta::any(true, false); // true
bool value4 = m3ta::any(true, 1, 2L, 4.0f, 9.0, 85.0L); // true

using Type1 = m3ta::AnyT<bool, true, true>; // std::integral_constant<bool, true>
using Type2 = m3ta::AnyT<bool, false, false>; // std::integral_constant<bool, false>
using Type3 = m3ta::AnyT<bool, true, false>; // std::integral_constant<bool, true>
using Type4 = m3ta::AnyT<bool, 1, 2, 4, 9, 85>; // std::integral_constant<bool, true>
```

Ending Occurrences

```
#include <m3ta/endingoccurrences>
```

Counts the number of occurrences of a value at the end of a sequence of elements.

Functions

m3ta::endingOccurrences

```
template<typename T_Search, typename ... T_Values>
constexpr std::size_t
endingOccurrences(T_Search search, T_Values ... values) noexcept
```

Template Parameters

- **T_Search** (automatically deduced) — Type of the value to search for.
- **T_Values** (automatically deduced) — Types of the values to search in.

Function Parameters

- **search** – Value to search for.
- **values** – Variable number of values to search in. Those can be of heterogeneous types.

Note: Values of heterogeneous types can be passed to the *values* parameter only if they all can be compared for equality (==) against the value to be searched for.

Returns

The number of occurrences of the *search* value at the end of the *values* elements.

Traits

m3ta::EndingOccurrences

```
template<typename T, T T_search, T ... T_values>
struct EndingOccurrences
```

Template Parameters

- **T** – Type of the values to check.
- **T_Search** – Value to search for.
- **T_Values** – Variable number of values to search in.

Member Types

type
The type `std::integral_constant<std::size_t, value>` where *value* is the result of the function *m3ta::endingOccurrences()*.

value_type
The type `std::size_t`.

Member Constants

static constexpr std::size_t value
The number of occurrences of the *search* value at the end of the *T_values* elements.

Aliases

m3ta::EndingOccurrencesT

```
template<typename T, T T_search, T ... T_values>
using EndingOccurrencesT =
    typename EndingOccurrences<T, T_search, T_values ...>::type;
```

Usage Examples

```
std::size_t value1 = m3ta::endingOccurrences(4, 1, 2, 4, 4, 4); // 3
std::size_t value2 = m3ta::endingOccurrences(4, 1, 4, 2, 4, 4); // 2
std::size_t value3 = m3ta::endingOccurrences(4.0f, std::complex<float>(4, 0)); // 1

using Type1 = m3ta::EndingOccurrencesT<int, 4, 1, 2, 4, 4, 4>; // std::integral_constant<std::size_t,
using Type2 = m3ta::EndingOccurrencesT<int, 4, 1, 4, 2, 4, 4>; // std::integral_constant<std::size_t,
```

Is Complex?

```
#include <m3ta/iscomplex>
```

Checks if a type is the same as `std::complex`.

Note: Following the behavior of the type traits from the C++ standard library, the check will return *false* is a `std::complex` type is passed as a reference.

Functions

m3ta::isComplex

```
template<typename T>
constexpr bool
isComplex() noexcept
```

Template Parameters

- **T** — Type to check.

Returns

Whether the type is the same as `std::complex`.

Traits

m3ta::IsComplex

```
template<typename T>
struct IsComplex
```

Template Parameters

- **T** – Type to check.

Member Types

type
The type `std::integral_constant<bool, value>` where *value* is the result of the function *m3ta::isComplex()*.

value_type
The type `bool`.

Member Constants

static constexpr bool value
Whether the type is the same as `std::complex`.

Aliases

m3ta::IsComplexT

```
template<typename T>
using IsComplexT = typename IsComplex<T>::type;
```

Usage Examples

```
bool value1 = m3ta::isComplex<float>() // false
bool value2 = m3ta::isComplex<std::complex<float>>() // true
bool value3 = m3ta::isComplex<std::complex<float> &>() // false

using Type1 = m3ta::IsComplexT<float>; // std::integral_constant<bool, false>
using Type2 = m3ta::IsComplexT<std::complex<float>>; // std::integral_constant<bool, true>
using Type3 = m3ta::IsComplexT<std::complex<float> &>; // std::integral_constant<bool, false>
```

Is Operator Callable?

```
#include <m3ta/isoperatorcallable>
```

Checks if an operator can be called on given types.

Enumerators

m3ta::UnaryOperator

```
enum class UnaryOperator {
    plus,
    minus,
    postfixIncrement,
    postfixDecrement,
    prefixIncrement,
    prefixDecrement,
    logicalNot,
```

```
    bitwiseNot
};
```

plus
Plus operator `+a`.

minus
Minus operator `-a`.

posfixIncrement
Postfix increment operator `a++`.

postfixDecrement
Postfix decrement operator `a--`.

prefixIncrement
Prefix increment operator `++a`.

prefixDecrement
Prefix decrement operator `--a`.

logicalNot
Logical NOT operator `!a`.

bitwiseNot
Bitwise NOT operator `~a`.

m3ta::BinaryOperator

```
enum class BinaryOperator {
    assignment,
    addition,
    subtraction,
    multiplication,
    division,
    modulo,
    equalTo,
    notEqualTo,
    greaterThan,
    lessThan,
    greaterThanOrEqualTo,
    lessThanOrEqualTo,
    logicalAnd,
    logicalOr,
    bitwiseAnd,
    bitwiseOr,
    bitwiseXor,
    bitwiseLeftShift,
    bitwiseRightShift,
    additionAssignment,
    subtractionAssignment,
    multiplicationAssignment,
    divisionAssignment,
    moduloAssignment,
    bitwiseAndAssignment,
    bitwiseOrAssignment,
    bitwiseXorAssignment,
    bitwiseLeftShiftAssignment,
    bitwiseRightShiftAssignment
};
```

assignment
Assignment operator `a = b`.

addition
Addition operator `a + b`.

subtraction
Subtraction operator `a - b`.

multiplication
Multiplication operator `a * b`.

division
Division operator `a / b`.

modulo
Modulo operator `a % b`.

equalTo
Equal to operator `a == b`.

notEqualTo
Not equal to operator `a != b`.

greaterThan
Greater than operator `a > b`.

lessThan
Less than operator `a < b`.

greaterThanOrEqualTo
Greater than or equal to operator `a >= b`.

lessThanOrEqualTo
Less than or equal to operator `a <= b`.

logicalAnd
Logical AND operator `a && b`.

logicalOr
Logical OR operator `a || b`.

bitwiseAnd
Bitwise AND operator `a & b`.

bitwiseOr
Bitwise OR operator `a | b`.

bitwiseXOr
Bitwise XOR operator `a ^ b`.

bitwiseLeftShift
Bitwise left shift operator `a << b`.

bitwiseRightShift
Bitwise right shift operator `a >> b`.

additionAssignment
Addition assignment operator `a += b`.

subtractionAssignment
Subtraction assignment operator `a -= b`.

multiplicationAssignment
Multiplication assignment operator `a *= b`.

divisionAssignment
Division assignment operator `a /= b`.

moduloAssignment
Modulo assignment operator `a %= b`.

bitwiseAndAssignment
Bitwise AND assignment operator `a &= b`.

bitwiseOrAssignment
Bitwise OR assignment operator `a |= b`.

bitwiseXOrAssignment
Bitwise XOR assignment operator `a ^= b`.

bitwiseLeftShiftAssignment
Bitwise left shift assignment operator `a <<= b`.

bitwiseRightShiftAssignment
Bitwise right shift assignment operator `a >>= b`.

Functions

m3ta::isOperatorCallable

```
template<UnaryOperator T_operator, typename T>
constexpr bool
isOperatorCallable() noexcept

template<BinaryOperator T_operator, typename T, typename T_Other>
constexpr bool
isOperatorCallable() noexcept
```

Template Parameters

- **T_operator** — Operator to check for.
- **T** — Type of the first operand.
- **T_Other** — Type of the second operand. For binary operators only.

Returns

Whether the operator can be called on the operand(s).

Traits

m3ta::IsOperatorCallable

```
template<typename T_Operator, T_Operator T_operator, typename ... T>
struct IsOperatorCallable;
```

Template Parameters

- **T_Operator** – Type of the operator to check for.
- **T_operator** – Operator to check for.
- **T** - Type of the operand(s). Only one operand is expected when checking against unary operators, and two for the binary operators.

Member Types

type
The type `std::integral_constant<bool, value>` where *value* is the result of the function *m3ta::isOperatorCallable()*.

value_type
The type `bool`.

Member Constants

static constexpr bool value
Whether the operator can be called on the operand(s).

Aliases

m3ta::IsOperatorCallableT

```
template<typename T_Operator, T_Operator T_operator, typename ... T>
using IsOperatorCallableT =
    typename IsOperatorCallable<T_Operator, T_operator, T ...>::type;
```

Usage Examples

```
bool value1 = m3ta::isOperatorCallable<
    m3ta::UnaryOperator::logicalNot,
    int
>(); // true
bool value2 = m3ta::isOperatorCallable<
    m3ta::BinaryOperator::addition,
    int,
    float
>(); // true
bool value3 = m3ta::isOperatorCallable<
    m3ta::BinaryOperator::multiplication,
    int,
    std::complex<float>
>(); // false

using Type1 = m3ta::IsOperatorCallableT<
    m3ta::UnaryOperator::logicalNot,
    int
>; // std::integral_constant<bool, true>
using Type2 = m3ta::IsOperatorCallableT<
```

```
    m3ta::BinaryOperator::addition,
    int,
    float
>; // std::integral_constant<bool, true>
using Type3 = m3ta::IsOperatorCallableT<
    m3ta::BinaryOperator::multiplication,
    int,
    std::complex<float>
>; // std::integral_constant<bool, false>
```

Maximum

```
#include <m3ta/maximum>
```

Returns the greater of two values.

See Also

Minimum.

Functions

m3ta::maximum

```
template<typename T>
constexpr T
maximum(T value1, T value2) noexcept
```

Template Parameters

- **T** (automatically deduced) — Type of the values to compare.

Function Parameters

- **value1** – First value to compare.
- **value2** – Second value to compare.

Returns

The greater of the two values.

Traits

m3ta::Maximum

```
template<typename T, T T_value1, T T_value2>
struct Maximum
```

Template Parameters

- **T** – Type of the values to check.

- **T_value1** – First value to compare.
- **T_value2** – Second value to compare.

Member Types

type
The type `std::integral_constant<T, value>` where *value* is the result of the function *m3ta::maximum()*.

value_type
The type `T`.

Member Constants

static constexpr T value
The greater of the two values.

Aliases

m3ta::MaximumT

```
template<typename T, T T_value1, T T_value2>
using MaximumT = typename Maximum<T, T_value1, T_value2>::type;
```

Usage Examples

```
auto value = m3ta::maximum(1, 2); // 2

using Type = m3ta::MaximumT<int, 1, 2>; // std::integral_constant<int, 2>
```

Minimum

```
#include <m3ta/minimum>
```

Returns the greater of two values.

See Also

Maximum.

Functions

m3ta::minimum

```
template<typename T>
constexpr T
minimum(T value1, T value2) noexcept
```

Template Parameters

- **T** (automatically deduced) — Type of the values to compare.

Function Parameters

- **value1** – First value to compare.
- **value2** – Second value to compare.

Returns

The greater of the two values.

Traits

m3ta::Minimum

```
template<typename T, T T_value1, T T_value2>
struct Minimum
```

Template Parameters

- **T** – Type of the values to check.
- **T_value1** – First value to compare.
- **T_value2** – Second value to compare.

Member Types

type
The type `std::integral_constant<T, value>` where *value* is the result of the function *m3ta::minimum()*.

value_type
The type `T`.

Member Constants

static constexpr T value
The greater of the two values.

Aliases

m3ta::MinimumT

```
template<typename T, T T_value1, T T_value2>
using MinimumT = typename Minimum<T, T_value1, T_value2>::type;
```

Usage Examples

```
auto value = m3ta::minimum(1, 2); // 1

using Type = m3ta::MinimumT<int, 1, 2>; // std::integral_constant<int, 1>
```

None

```
#include <m3ta/none>
```

Checks if none of the values evaluate to *true*.

See Also

All and *Any*.

Functions

m3ta::none

```
template<typename ... T_Values>
constexpr bool
none(T_Values ... values) noexcept
```

Template Parameters

- **T_Values** (automatically deduced) — Types of the values to check.

Function Parameters

- **values** – Variable number of values to check. Those can be of heterogeneous types.

Returns

Whether none of the values evaluate to *true*.

Traits

m3ta::None

```
template<typename T, T ... T_values>
struct None
```

Template Parameters

- **T** – Type of the values to check.
- **T_values** – Variable number of values to check.

Member Types

type
The type `std::integral_constant<bool, value>` where *value* is the result of the function *m3ta::none()*.

value_type
The type `bool`.

Member Constants

static constexpr bool value
Whether none of the values evaluate to *true*.

Aliases

m3ta::NoneT

```
template<typename T, T ... T_values>
using NoneT = typename None<T, T_values ...>::type;
```

Usage Examples

```
bool value1 = m3ta::none(true, true); // false
bool value2 = m3ta::none(false, false); // true
bool value3 = m3ta::none(true, false); // false
bool value4 = m3ta::none(true, 1, 2L, 4.0f, 9.0, 85.0L); // false

using Type1 = m3ta::NoneT<bool, true, true>; // std::integral_constant<bool, false>
using Type2 = m3ta::NoneT<bool, false, false>; // std::integral_constant<bool, true>
using Type3 = m3ta::NoneT<bool, true, false>; // std::integral_constant<bool, false>
using Type4 = m3ta::NoneT<bool, 1, 2, 4, 9, 85>; // std::integral_constant<bool, false>
```

Power

```
#include <m3ta/power>
```

Functions

m3ta::power

```
template<typename T_Base, typename T_Exponent>
constexpr MultiplicationResultT<T_Base, T_Base>
power(T_Base base, T_Exponent exponent) noexcept
```

Template Parameters

- **T_Base** (automatically deduced) — Type of the base value.

- **T_Exponent** (automatically deduced) — Type of the exponent value.

Function Parameters

- **base** – Base value.
- **exponent** – Exponent value.

Returns

The number *base* raised to the power *exponent*.

Traits

m3ta::Power

```
template<typename T, T T_base, T T_exponent>
struct Power
```

Template Parameters

- **T** – Type of the values.
- **T_base** – Base value.
- **T_exponent** – Exponent value.

Member Types

type
The type `std::integral_constant<T, value>` where *value* is the result of the function *m3ta::power()*.

value_type
The type `T`.

Member Constants

static constexpr T value
The number *T_base* raised to the power *T_exponent*.

Aliases

m3ta::PowerT

```
template<typename T, T T_base, T T_exponent>
using PowerT = typename Power<T, T_base, T_exponent>::type;
```

Usage Examples

```
auto value1 = m3ta::power(4, 3); // 64
auto value2 = m3ta::power(1.0f, -1); // 0.5
auto value3 = m3ta::power(std::complex<float>(1, 1), 2); // std::complex<float>(0, 2)

using Type = m3ta::PowerT<int, 4, 3>; // std::integral_constant<int, 64>
```

Product

```
#include <m3ta/product>
```

Computes the product of a sequence of elements.

The computation follows a left-to-right associativity, meaning that `a * b * c` is evaluated as `(a * b) * c`.

See Also

The Nested Initializers Trick for Multidimensional Arrays.

Functions

m3ta::product

```
template<typename T_First, typename ... T_Others>
constexpr m3ta::MultiplicationResultT<T_First, T_Others ...>
product(T_First first, T_Others ... others) noexcept
```

The arguments passed do not have to be of numeric type—custom types such as matrices and vectors can be passed as soon as they define arithmetic multiplications with the values preceding and/or following them.

Template Parameters

- **T_First** (automatically deduced) — Type of the first value to multiply.
- **T_Others** (automatically deduced) — Types of the other values to mutiply.

Function Parameters

- **first** – First value to multiply.
- **others** – Other values to multiply.

Returns

The product of the values.

Traits

m3ta::Product

```
template<typename T, T T_first, T ... T_others>
struct Product
```

Template Parameters

- **T** – Type of the values.
- **T_first** – First value to multiply.
- **T_others** – Other values to multiply.

Member Types

type
The type `std::integral_constant<T, value>` where *value* is the result of the function *m3ta::product()*.

value_type
The type `T`.

Member Constants

static constexpr T value
The product of the values.

Aliases

m3ta::ProductT

```
template<typename T, T T_first, T ... T_others>
using ProductT = typename Product<T, T_first, T_others ...>::type;
```

m3ta::IndexProduct

```
template<std::size_t T_first, std::size_t ... T_others>
using IndexProduct = Product<std::size_t, T_first, T_others ...>;
```

m3ta::IndexProductT

```
template<std::size_t T_first, std::size_t ... T_others>
using IndexProductT = typename IndexProduct<T_first, T_others ...>::type;
```

Usage Examples

```
auto value1 = m3ta::product(1, 2, 4); // 8
auto value2 = m3ta::product(4.9, 85); // 416.5
auto value3 = m3ta::product(std::complex<float>(2, 2), 4); // std::complex<float>(8, 8)

using Type = m3ta::ProductT<int, 1, 2, 4>; // std::integral_constant<int, 8>
```

2.1.2 Types

Traits defining a `type` member holding the resulting type of an operation.

An alias ending with the prefix *T* that refers to the `type` member is also available as a convenience.

Arithmetic Operation Result

```
#include <m3ta/arithmeticoperationresult>
```

Deduces the resulting type of an arithmetic operation between the given type elements.

The deduction follows a left-to-right associativity, meaning that `a + b + c` is evaluated as `(a + b) + c`.

Enumerators

m3ta::ArithmeticOperator

```
enum class ArithmeticOperator {
    addition,
    subtraction,
    multiplication,
    division
};
```

addition
: Addition operator `+`.

subtraction
: Subtraction operator `-`.

multiplication
: Multiplication operator `*`.

division
: Division operator `/`.

Traits

m3ta::ArithmeticOperationResult

```
template<ArithmeticOperator T_operator, typename ... T_Values>
struct ArithmeticOperationResult
```

Template Parameters

- **T_operator** – Operator to use for the check.
- **T_Values** – Variable number of types to check.

Member Types

type
: The resulting type of the arithmetic operation between the given type elements.

Aliases

m3ta::ArithmeticOperationResultT

```
template<ArithmeticOperator T_operator, typename ... T_Values>
using ArithmeticOperationResultT =
    typename ArithmeticOperationResult<T_operator, T_Values ...>::type;
```

m3ta::AdditionResult

```
template<typename ... T_Values>
using AdditionResult =
    ArithmeticOperationResult<ArithmeticOperator::addition, T_Values ...>;
```

m3ta::AdditionResultT

```
template<typename ... T_Values>
using AdditionResultT =
    typename AdditionResult<T_Values ...>::type;
```

m3ta::SubtractionResult

```
template<typename ... T_Values>
using SubtractionResult =
    ArithmeticOperationResult<ArithmeticOperator::subtraction, T_Values ...>;
```

m3ta::SubtractionResultT

```
template<typename ... T_Values>
using SubtractionResultT =
    typename SubtractionResult<T_Values ...>::type;
```

m3ta::MultiplicationResult

```
template<typename ... T_Values>
using MultiplicationResult =
    ArithmeticOperationResult<ArithmeticOperator::multiplication, T_Values ...>;
```

m3ta::MultiplicationResultT

```
template<typename ... T_Values>
using MultiplicationResultT =
    typename MultiplicationResult<T_Values ...>::type;
```

m3ta::DivisionResult

```
template<typename ... T_Values>
using DivisionResult =
    ArithmeticOperationResult<ArithmeticOperator::division, T_Values ...>;
```

m3ta::DivisionResultT

```
template<typename ... T_Values>
using DivisionResultT =
    typename DivisionResult<T_Values ...>::type;
```

Usage Examples

```
using Type1 = m3ta::AdditionResultT<char, short>; // int
using Type2 = m3ta::SubtractionResultT<int, float>; // float
using Type3 = m3ta::ArithmeticOperationResultT<
    m3ta::ArithmeticOperator::multiplication,
    float,
    std::complex<float>
>; // std::complex<float>
```

Concatenate Integer Sequences

```
#include <m3ta/concatenateintegersequences>
```

Concatenates two integer sequences into one.

Traits

m3ta::ConcatenateIntegerSequences

```
template<typename T_Sequence1, typename T_Sequence2>
struct ConcatenateIntegerSequences
```

Template Parameters

- **T_Sequence1** – The first integer sequence.
- **T_Sequence2** – The second integer sequence.

Member Types

type
The type *m3ta::IntegerSequence* resulting of the concatenation.

Aliases

m3ta::ConcatenateIntegerSequencesT

```
template<typename T_Sequence1, typename T_Sequence2>
using ConcatenateIntegerSequencesT =
    typename ConcatenateIntegerSequences<T_Sequence1, T_Sequence2>::type;
```

Usage Examples

```
using Type = m3ta::ConcatenateIntegerSequencesT<
    m3ta::IntegerSequence<int, 1, 2>,
    m3ta::IntegerSequence<int, 3, 4>
>; // m3ta::IntegerSequence<int, 1, 2, 3, 4>
```

Extract Integer Sequence

```
#include <m3ta/extractintegersequence>
```

Extracts the integer sequence as an object instance, such as a *std::array*.

Traits

m3ta::ExtractIntegerSequence

```
template<typename T_Sequence>
struct ExtractIntegerSequence
```

Template Parameters

- **T_Sequence** – Integer sequence.

Member Functions

std::array<T, sizeof ... (T_values)> asArray()
The integer sequence as a `std::array`.

Usage Examples

```
auto array = m3ta::ExtractIntegerSequence<
    m3ta::IndexSequence<0, 1, 2, 4>
>::asArray(); // std::array<std::size_t, 4>
```

Nested Initializer Lists

```
#include <m3ta/nestedinitializerlists>
```

A given type wrapped within a number of nested `std::initializer_list`.

In other words, `m3ta::NestedInitializerListsT<int, 2>` is the same as `std::initializer_list<std::initializer_list<int>>`.

See Also

The Nested Initializers Trick for Multidimensional Arrays.

Traits

m3ta::NestedInitializerLists

```
template<typename T, std::size_t T_levels>
struct NestedInitializerLists
```

Template Parameters

- **T** – Inner type.
- **T_levels** – Number of `std::initializer_list` wrapping the type `T`.

Member Types

type
The type of the outer `std::initializer_list`. Returns `T` if *T_levels* is 0.

Aliases

m3ta::NestedInitializerListsT

```
template<typename T, std::size_t T_levels>
using NestedInitializerListsT =
    typename NestedInitializerLists<T, T_levels>::type;
```

Usage Examples

```
using Type1 = m3ta::NestedInitializerListsT<int, 0>; // int
using Type2 = m3ta::NestedInitializerListsT<int, 1>; // std::initializer_list<int>
using Type3 = m3ta::NestedInitializerListsT<int, 2>; // std::initializer_list<std::initializer_list<
```

Pass

```
#include <m3ta/pass>
```

Does nothing but passing the given template argument while taking an optional parameter pack.

This is a convenient way to test a SFINAE expression while always returning the required type.

See Also

Overload Resolution Using SFINAE.

Traits

m3ta::Pass

```
template<typename T, typename ... T_Dummies>
struct Pass
```

Template Parameters

- **T** – Type to pass.
- **T_Dummies** – Allows for some expressions to be ran through.

Member Types

type
The type `T`.

Aliases

m3ta::PassT

```
template<typename T, typename ... T_Dummies>
using PassT = typename Pass<T, T_Dummies ...>::type;
```

Usage Examples

```
using Type1 = m3ta::PassT<void, int, float>; // void
using Type2 = m3ta::PassT<short, int, float>; // short
```

Pop Integer Sequence

```
#include <m3ta/popintegersequence>
```

Removes a given number of elements from the end of an integer sequence.

Traits

m3ta::PopIntegerSequence

```
template<std::size_t T_count, typename T_Sequence>
struct PopIntegerSequence;
```

Template Parameters

- **T_count** – Number of elements to remove from the end.
- **T_Sequence** – The integer sequence.

Member Types

type
The type *m3ta::IntegerSequence* resulting of the elements removal.

Aliases

m3ta::PopIntegerSequenceT

```
template<std::size_t T_count, typename T_Sequence>
using PopIntegerSequenceT =
    typename PopIntegerSequence<T_count, T_Sequence>::type;

template<std::size_t T_count, typename T, T ... T_values>
using PopIntegerPack = PopIntegerSequence<
    T_count,
    IntegerSequence<T, T_values...>
>;

template<std::size_t T_count, typename T, T ... T_values>
using PopIntegerPackT =
    typename PopIntegerPack<T_count, T, T_values ...>::type;

template<std::size_t T_count, std::size_t ... T_values>
using PopIndexPack =
    PopIntegerSequence<
        T_count,
        IndexSequence<T_values...>
    >;

template<std::size_t T_count, std::size_t ... T_values>
using PopIndexPackT =
    typename PopIndexPack<T_count, T_values ...>::type;
```

Usage Examples

```
using Type1 = m3ta::PopIndexPackT<
    1,
    0, 1, 2, 4
>; // m3ta::IntegerSequence<std::size_t, 0, 1, 2>
using Type2 = m3ta::PopIntegerSequenceT<
    2,
    m3ta::IntegerSequence<int, 0, 1, 2, 4>
>; // m3ta::IntegerSequence<int, 0, 1>
```

Read-Only Parameter

```
#include <m3ta/readonlyparameter>
```

Resolves the most optimal read-only parameter signature for a given type.

In other words, it returns either `T` or `const T&` depending on the size of a type `T`.

Note: Any `const`, `volatile` and reference qualifiers are ignored during the resolution. As such, the types `T`, `T &&`, `const T &`, and so on, will all resolve to the same type.

Traits

m3ta::ReadOnlyParameter

```
template<typename T>
struct ReadOnlyParameter
```

Template Parameters

- **T** – Type to resolve.

Member Types

type
The most optimal read-only parameter signature for the type `T`.

Aliases

m3ta::ReadOnlyParameterT

```
template<typename T>
using ReadOnlyParameterT = typename ReadOnlyParameter<T>::type;
```

Usage Examples

```
using Type1 = m3ta::ReadOnlyParameterT<char>; // char
using Type2 = m3ta::ReadOnlyParameterT<std::complex<double>>; // const std::complex<double> &
using Type3 = m3ta::ReadOnlyParameterT<double *>; // double*
using Type4 = m3ta::ReadOnlyParameterT<double[32]>; // double[32]
using Type5 = m3ta::ReadOnlyParameterT<const volatile char &>; // char
using Type6 = m3ta::ReadOnlyParameterT<char &&>; // char
```

Remove Qualifiers

```
#include <m3ta/removequalifiers>
```

Removes any qualifiers from a given type.

Any `const`, `volatile` and reference qualifiers are removed. As such, the types `T`, `T &&`, `const T &`, and so on, will all resolve to `T`.

Arrays and pointers are preserved.

Traits

m3ta::RemoveQualifiers

```
template<typename T>
struct RemoveQualifiers
```

Template Parameters

- **T** – Type to remove the qualifiers from.

Member Types

type
The type `T` without any qualifier.

Aliases

m3ta::RemoveQualifiersT

```
template<typename T>
using RemoveQualifiersT = typename RemoveQualifiers<T>::type;
```

Usage Examples

```
using Type1 = m3ta::RemoveQualifiersT<int>; // int
using Type2 = m3ta::RemoveQualifiersT<const volatile int &>; // int
using Type3 = m3ta::RemoveQualifiersT<int &&>; // int
using Type4 = m3ta::RemoveQualifiersT<int *&>; // int *
using Type5 = m3ta::RemoveQualifiersT<int[32]>; // int[32]
```

Reverse Integer Sequence

```
#include <m3ta/reverseintegersequence>
```

Reverses an integer sequence.

Traits

m3ta::ReverseIntegerSequence

```
template<typename T_Sequence>
struct ReverseIntegerSequence;
```

Template Parameters

- **T_Sequence** – Integer sequence.

Member Types

type
The type *m3ta::IntegerSequence* resulting of the reversal.

Aliases

m3ta::ReverseIntegerSequenceT

```
template<typename T_Sequence>
using ReverseIntegerSequenceT =
    typename ReverseIntegerSequence<T_Sequence>::type;

template<typename T, T ... T_values>
using ReverseIntegerPack =
    ReverseIntegerSequence<IntegerSequence<T, T_values...>>;

template<typename T, T ... T_values>
using ReverseIntegerPackT =
    typename ReverseIntegerPack<T, T_values...>::type;

template<std::size_t ... T_values>
using ReverseIndexPack =
    ReverseIntegerSequence<IndexSequence<T_values...>>;

template<std::size_t ... T_values>
using ReverseIndexPackT =
    typename ReverseIndexPack<T_values...>::type;
```

Usage Examples

```
using Type1 = m3ta::ReverseIndexPackT<
   0, 1, 2, 4
 >; // m3ta::IntegerSequence<std::size_t, 4, 2, 1, 0>
using Type2 = m3ta::ReverseIntegerSequenceT<
    m3ta::IntegerSequence<int, 0, 1, 2, 4>
>; // m3ta::IntegerSequence<int, 4, 2, 1, 0>
```

2.1.3 Specials

For the uncategorizable ones.

Dependent Bool

```
#include <m3ta/dependentbool>
```

Creates a dependency to some dummy types and returns a *bool* value.

See Also

Static Assert.

Functions

m3ta::dependentBool

```
template<typename ... T_Dummies>
constexpr bool
dependentBool(bool value) noexcept
```

Template Parameters

- **T_Dummies** — Types to trigger the dependency.

Function Parameters

- **value** – *bool* value to return.

Returns

The *bool* value passed as argument.

Usage Examples

```
bool value1 = m3ta::dependentBool<int, float>(true); // true
bool value2 = m3ta::dependentBool<int, float>(false); // false
```

Integer Sequence

```
#include <m3ta/integersequence>
```

Sequence of integers.

This is the equivalent of C++14's `std::integer_sequence`.

Traits

`m3ta::IntegerSequence`

```
template<typename T, T ... T_values>
struct IntegerSequence
```

Template Parameters

- **T** – Type of the integer values.
- **T_values** – Integral values.

Member Types

`value_type`
The type `T`.

Member Functions

`std::size_t size() noexcept`
Number of values.

Aliases

m3ta::IndexSequence

```
template<std::size_t ... T_values>
using IndexSequence = IntegerSequence<std::size_t, T_values ...>;
```

Parameter Pack

#include <m3ta/parameterpack>

Pack of parameters.

Traits

m3ta::ParameterPack

```
template<typename ... T>
struct ParameterPack
```

Template Parameters

- **T** – Types to pack.

CHAPTER 3

Additional Information

3.1 FAQ

3.1.1 Why providing constexpr functions when traits would be enough?

Functions have a friendlier syntax and can be more flexible to use. Indeed, variadic functions accept a variable number of heterogeneous arguments while their trait counterpart only accept the values passed to the variadic non-type template parameters that can be implicitely converted to a same type. Furthermore, the non-type template parameters allowed are restricted to a smaller subset which do not include floating-points or custom types.

```
m3ta::all(true, 2, 4.0f); //< Valid.
m3ta::AllT<bool, true, 2, 4>::value; //< Valid, implicit conversions of int to bool.
m3ta::AllT<bool, true, 2, 4.0f>::value; //< Invalid due to the use of a float.
```

3.1.2 Why also providing traits on top of existing functions then?

Because functions can only return values, not types. Retrieving a constant value might be enough in most cases but not when used with the tag dispatching technique or in some SFINAE contexts where a type is expected to resolve which overload to use. Hence why each existing function is being wrapped into a trait inheriting from `std::integral_constant`.

See Also

Tag Dispatching.

3.2 Contributing

Found a bug or got a feature request? Don't keep it for yourself, log a new issue on the GitHub project page.

3.3 Changes

3.3.1 v0.0.2 (2015-03-21)

- Add the *ReverseIntegerSequence* and *ExtractIntegerSequence* traits.

- Add aliases for integer parameter packs, such as *PopIntegerPackT*.
- Swapped the position of the *T_Sequence* and *T_count* template arguments for the *PopIntegerSequence* trait.
- Minor fixes.

3.3.2 v0.0.1 (2015-01-09)

- Initial release.

3.4 Versioning

Versions numbers will comply with the Sementic Versioning (SemVer) specification and will be written in the form `<major>.<minor>.<patch>`, where:

- backwards incompatible API changes increments the major version (and resets the minor and patch versions).
- backwards compatible API additions/changes increments the minor version (and resets the patch version).
- bug fixes not affecting the API increments the patch version.

3.5 License

CHAPTER 4

Indices

- *genindex*
- *modindex*