

Building Blocks of Metaprogramming

Taming the Dragons of C++'s built in code generator

Immer eine zündende Idee.

Contact: holmes@auto-intern.de






Overview

- 📊 what problems does metaprogramming solve / cause?
- 📊 when should we use it?
- 📊 what are common stumbling blocks?
- 📊 what's in our toolbox?
- 📊 constexpr vs. template functional programming
- 📊 real world example

What problems does metaprogramming solve?

Different tools solve different things:

-  anything that results in a value should be done with `constexpr`
-  anything that results in a type should be done with templates
-  anything that is meant to sabotage a codebase should be done with macros

Resulting in a type means code generation depending on input types

```
template<typename T>
void fImpl(T a, std::true_type);
template<typename T>
void fImpl(T a, std::false_type);
template<typename T>
f(T a) {
    fImpl(a, DeciderMetaFunction<T>{});
}
```

```
template<typename T>
f(T a) {
    WhatToDo<T>::f(input);
    WhatToDo<T>{}(input);
}
```

Evaluation at compile time elevates the level of programming

```
//datasheet says set bits 10:2 and bit 0  
#define MY_NAMED_MASK 0x03FD  
constexpr auto myNamedMask = "10:2,0"_mask;
```

 The programmer can express themselves in a manor closer to their mental model

```
void foo("10:2,0"_mask);  
void foo(MY_NAMED_MASK);  
std::tie(a, b, c) = f(x);
```

Allows static checking

```
#define TEMPERATURE_A 1
#define TEMPERATURE_B 0
void f(unsigned long degrees, int b = 0);
void f(int * a);
void g(bool on, int degrees);

f(TEMPERATURE_A); //probably no surprises
f(TEMPERATURE_B); //will not pick the overload you
think

g(TEMPERATURE_A, false); //no error!!!!

//*stay tuned for solution
```

Allows lazy evaluation

```
struct matrix_add;

matrix_add operator +(matrix const& a, matrix const& b) {
    return matrix_add{a, b};
}

struct matrix_add {
    matrix_add(matrix const& a, matrix const& b) : a{a}, b{b} { }

    operator matrix() const {
        matrix result;
        // Do the addition.
        return result;
    }
private:
    matrix const& a, b;
};
```

Encapsulation of expertise

- 📡 How does `std::tie` work?
- 📡 What optimization is used in `std::find()` with random access char iterators?
- 📡 How is alignment guaranteed in `std::tuple`?

I don't know! I don't have to know, its encapsulated.

What problems does metaprogramming cause?

- 🔴 Ugly, scary compiler errors
- 🔴 Code that is impossible for novice users to understand






"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

-Brian W. Kernighan and P. J. Plauger in The Elements of Programming Style.

Template errors are scary



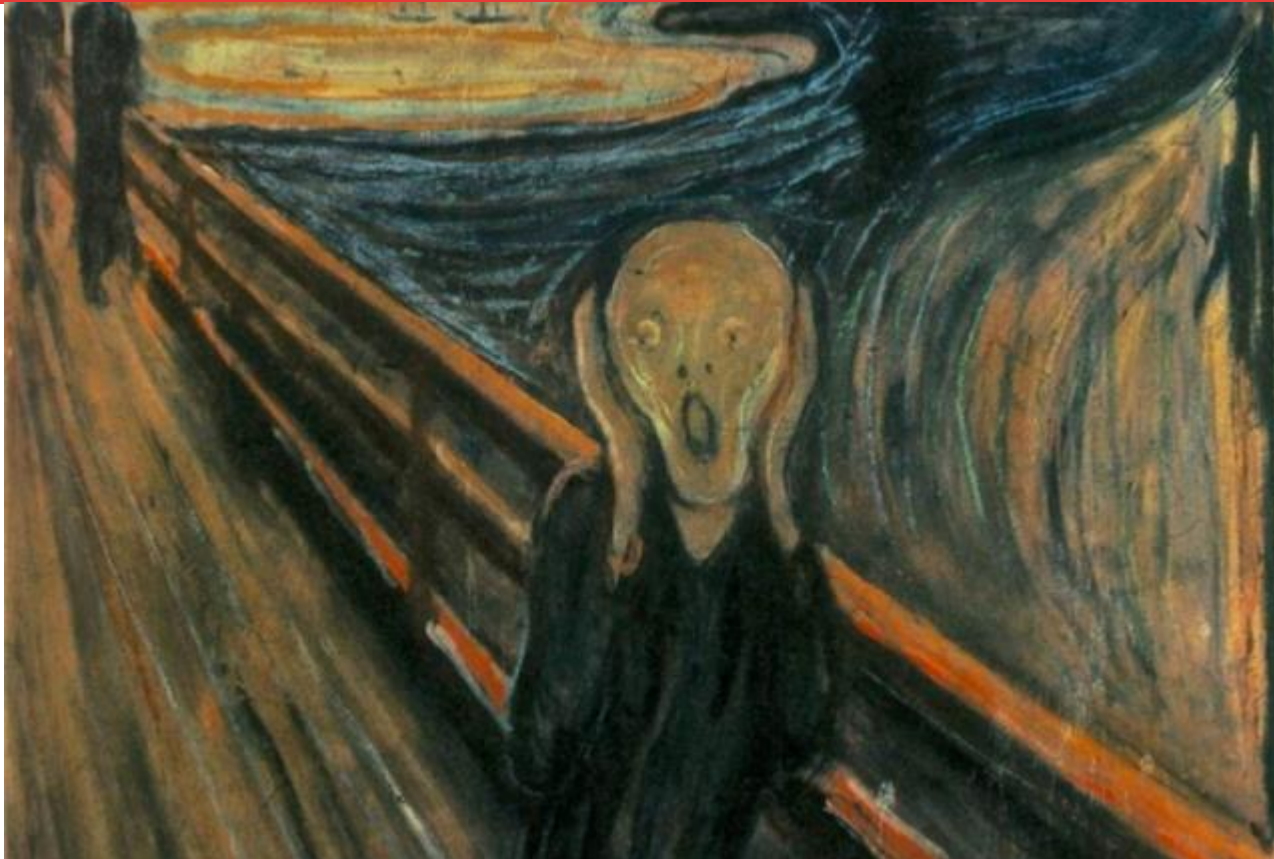
When should we use it?

-  In library code
-  Only when super unit tested
-  Only when solving an generic problem with a provable solution
-  Never in a one off solution
-  Never in code that other people need to understand

What are common stumbling blocks?

- 📡 Functional programming style
- 📡 Most idioms are just different use cases of template specialization
- 📡 Syntax can be cryptic (stay tuned)
- 📡 Loops, iteration etc. is all implemented using recursion

Beware of slide 13, the horror begins



Templates

```
template<typename T, int I>
struct Array {
    T data_[I];
    // implementation here
};
```

```
template<typename T>
T square(T in) {
    return in*in;
}
```

Using

```
typedef std::vector<int> intVec;
```

```
using intVec2 = std::vector<int>;
```

```
template<unsigned I>
```

```
using intArray = std::array<int, I>;
```

```
intArray<4> ia;
```

Variadic templates

```
template<typename... Ts>
void myPrintf(std::string s, Ts...args) {
    printf(s.c_str(), args...);
}
```

```
template<typename... Ts>
struct S : Ts... {};
```


Data storage

```
struct S {  
    friend int getI(const S& s);  
    friend bool getB(const S& s);  
    S(int i, bool b):i_{i},b_{b} {}  
private:  
    int i_;  
    bool b_;  
};  
int getI(const S& s){ return s.i_;}  
bool getB(const S& s){ return s.b_;}  
  
S myS(4, false);  
auto i = getI(myS);
```

```
template<int I, bool B>  
struct S {};  
  
template<typename T>  
struct GetI;  
template<int I, bool B>  
struct GetI<S<I, B>> {  
    static constexpr int value = I;  
}  
  
using MyS = S<4, false>;  
auto i = GetI<MyS>::value;
```

Composition

```
template<typename T, bool B>
struct S2 {};

using myS2 = S2<MyS, false>;

template<int I, bool B, bool B2>
struct GetI<S2<S<I, B>, B2>> {
    static constexpr int value = I;
}
```

If / switch

```
int f(int i, bool b) {  
    if (b) {  
        switch (i) {  
            default:  
                return 99;  
            case 42:  
                return 1  
        };  
    }  
    return 22;  
}
```

```
int i = f(42);
```

```
template<int I, bool B>  
struct F {  
    static constexpr int value = 22;  
};
```

```
template<int I>  
struct F<I, true> {  
    static constexpr int value = 99;  
};
```

```
template<>  
struct F<42, true> {  
    static constexpr int value = 1;  
};
```

```
int i = F<42>::value;
```

Partial specialization

```
template<int I, bool B>
struct S{}; //default
template<int I> //specialization 1
struct S<I, false> {
    static constexpr int value = 1;
};
template<int I> //specialization 2
struct S<I, true> {
    static constexpr int value = 2;
};
template<bool B> //specialization 3
struct S<4, B> {
    static constexpr int value = 3;
};
template<> //specialization 4
struct S<4, false> {
    static constexpr int value = 4;
};
```

```
int a = S<5, false>::value;
//^is 1
```

```
int b = S<4, false>::value;
//^is 4
```

```
int c = S<4, true>::value;
//^error, 3 and 2 are
equally specialized and
both match
```

```
int d = S<9, true>::value;
//^is 2
```

Containers / loops

```
template<typename...Ts>  
struct list {};
```

```
template<int I, typename T>  
struct at_impl;  
template<int I, typename T, typename...Ts>  
struct at_impl<I, list<T, Ts...>> : at_impl<I - 1, list<Ts...>>{};  
template<typename T, typename... Ts>  
struct at_impl<0, list<T, Ts...>> { using type = T; };
```




```
template<int I, typename T>  
using at = typename at_impl<I, T>::type;
```

```
using L = list<int, bool, float, bool>;
```

```
at<2, L> f = 1.4;
```

Value and type wrappers

A parameter can be a:

-  type
-  value of an integral type (int, bool, pointer, enum etc.)
-  template

There is no polymorphism or function overloading. The solution is wrapping values and templates in type wrappers.

Value and type wrappers 2

```
template<typename T, T V>
struct Integral {
    static constexpr T value = V;
};
using T = Integral<int, 4>;
auto i = T::value;

template<template<typename...> class T>
struct Template {
    template<typename...Ts>
    using apply = T<Ts...>;
};

template<typename...Ts> struct list {};
std::vector<std::any> v;

template<typename T> struct function {};
void f(const std::any& a);
```

Traits

```
template<typename T>
struct MyTraits {
    using type = DefaultType;
};

//somewhere else
template<>
struct MyTraits<MyType> {
    using type = MyTypesTraitsType;
};

//in user code
int i = MyTraits<MyType>::type::value;
```


Point of instantiation

For a function template specialization, a member function template specialization, or a specialization for a member function or static data member of a class template, if the specialization is implicitly instantiated because it is referenced from within another template specialization and the context from which it is referenced depends on a template parameter, the point of instantiation of the specialization is the point of instantiation of the enclosing specialization. Otherwise, the point of instantiation for such a specialization immediately follows the namespace scope declaration or definition that refers to the specialization.

If a function template or member function of a class template is called in a way which uses the definition of a default argument of that function template or member function, the point of instantiation of the default argument is the point of instantiation of the function template or member function specialization.

For a class template specialization, a class member template specialization, or a specialization for a class member of a class template, if the specialization is implicitly instantiated because it is referenced from within another template specialization, if the context from which the specialization is referenced depends on a template parameter, and if the specialization is not instantiated previous to the instantiation of the enclosing template, the point of instantiation is immediately before the point of instantiation of the enclosing template. Otherwise, the point of instantiation for such a specialization immediately precedes the namespace scope declaration or definition that refers to the specialization.

Point of instantiation 2

```
template<typename T>
struct ST { using type = int; };

template<typename T>
struct S {
    int f() {
        return typename ST<T>::type(3.14) * 9;
    }
};

template<>
struct ST<float> { using type = float; };

int foo() {
    return S<float>{}.f();
}
```

Point of instantiation 3

```
template<typename T, int N>  
struct Pointerify {  
    using type = typename Pointerify<T,N-1>::type*;  
};
```

```
template<typename T>  
struct Pointerify<T, 0> {  
    using type = T;  
};
```

```
using P = Pointerify<int, 2>::type;
```

Constexpr functional programming

```
constexpr int max(int i) {  
    return i;  
}  
  
template<typename...T>  
constexpr int max(int i, int j, T...args) {  
    return max(i>j ? i : j, args...);  
}  
  
char buf[max(1, 5, 9, 3, 1)];
```

Enforcing invariants

```
struct time { double value_; };
struct distance { double value_; };

time operator+(time lhs, time rhs) {
    return time{ lhs.value_ + rhs.value_ };
}
distance operator+(distance lhs, distance rhs) {
    return distance{ lhs.value_ + rhs.value_ };
}
??? operator/(distance lhs, time rhs) {
    return ??? {lhs.value_ / rhs.value_};
}
```

Enforcing invariants 2

```
struct speed { double value_; };
```

```
speed operator/(distance lhs, time rhs) {  
    return speed {lhs.value_ / rhs.value_};  
}
```

```
acceleration operator/(speed lhs, time rhs);  
area operator*(distance, distance);  
volume operator*(distance, area);
```

```
??? operator*(area, speed);
```

Enforcing invariants 3

```
template<int...D>
struct Unit { double value_; };

template<int...Is>
Unit<Is...> operator+(Unit<Is...> lhs, Unit<Is...> rhs) {
    return Unit<Is...> {lhs.value_ + rhs.value_};
}

template<int...Is, int...Js>
auto operator*(Unit<Is...> lhs, Unit<Js...> rhs) {
    return Unit<(Is + Js)...> {lhs.value_ * rhs.value_};
}

template<int...Is, int...Js>
auto operator/(Unit<Is...> lhs, Unit<Js...> rhs) {
    return Unit<(Is - Js)...> {lhs.value_ / rhs.value_};
}
```

static_assert / compile time errors

```
template<int...Is, int...Js>
auto operator*(Unit<Is...> lhs, Unit<Js...> rhs) {
    static_assert(sizeof...(Is)==sizeof...(Js),"incompatible unit systems");
    return Unit<(Is + Js)...> {lhs.value_ * rhs.value_};
}

constexpr int foo(int i, int j) {
    return i>j? i+j: throw std::logic_error(" i must be greater than j ");
}

int assert_failed() {
    static int i = 5;
    return i++;
}

constexpr int crude_assert(bool b) {
    return b == true ? 0 : assert_failed();
}

constexpr int foo(int i, int j) {
    return crude_assert(i>j), i + j;
}
```


Compile time algorithms

```
//Sort
template<typename TList, typename TPred = LessP>
struct SortImpl {
    static_assert(AlwaysFalse<TList>::value, "implausible type");
};
//empty input case
template<template<typename, typename > class TPred>
struct SortImpl<list<>, Template<TPred>> {
    using type = list<>;
};
//one element case
template<typename T, template<typename, typename > class TPred>
struct SortImpl<list<T>, Template<TPred>> {
    using type = list<T>;
};
//two or more elements case
template<typename ... Ts, template<typename, typename > class TPred>
struct SortImpl<list<Ts...>, Template<TPred>> : Detail::Sort<list<>, TPred, Ts...> {};
//alias
template<typename TList, typename TPred = LessP>
using Sort = typename SortImpl<TList, TPred>::type;
```

Compile time algorithms 2

```
namespace Detail {
    template<typename Out, template<typename, typename> class Pred, typename In,
        bool Tag, typename ... Ts>
    struct SortInsert;
    //next is less than insert, next is not end
    template<typename ... Os, template<typename, typename > class Pred, typename In,
        typename T1, typename T2, typename ... Ts>
    struct SortInsert<list<Os...>, Pred, In, true, T1, T2, Ts...> :
        SortInsert<list<Os..., T1>, Pred, In, Pred<T2, In>::value, T2, Ts...> {};
    //next is less than insert, next is end, terminate
    template<typename ... Os, template<typename, typename > class Pred, typename In,
        typename ... Ts>
    struct SortInsert<list<Os...>, Pred, In, true, Ts...> {
        using type = list<Os..., Ts..., In>;
    };
    //next is not less than insert, terminate
    template<typename ... Os, template<typename, typename > class Pred,
        typename In, typename ... Ts>
    struct SortInsert<list<Os...>, Pred, In, false, Ts...> {
        using type = list<Os..., In, Ts...>;
    };
}
```

Compile time algorithms 3

```
template<typename Out, template<typename, typename > class P, typename ... Ts>
struct Sort {
    static_assert(AlwaysFalse<TOut>::value, "implausible parameters");
};
//out and in are not empty
template<typename O, typename ... Os, template<typename, typename > class Pred,
    typename In, typename ... Ts>
struct Sort<list<O, Os...>, Pred, In, Ts...> : Sort< typename SortInsert<list<>,
    Pred, In, Pred<O, In>::value, O, Os...>::type, Pred, Ts...>{};
//out is empty, in is not empty
template<typename ... Os, template<typename, typename > class Pred,
    typename In, typename ... Ts>
struct Sort<list<Os...>, Pred, In, Ts...> :
    Sort<typename SortInsert<list<>, Pred, In, false, Os...>::type, Pred, Ts...>{};
//in is empty
template<typename ... Os, template<typename, typename > class P, typename ... Ts>
struct Sort<list<Os...>, P, Ts...> {
    using type = list<Os...>;
};
}
```

Brigand to the rescue

- 🔴 Brigand is a light-weight, fully functional, instant-compile time C++ 11 meta-programming library.
- 🔴 Similar to the Boost.MPL but modern
- 🔴 clang 3.4 and upward supported
- 🔴 GCC 4.8 and upward supported
- 🔴 Visual Studio 2013 Update 4 and upward supported (even though it sucks)