Template Metaprogramming

In this lesson, we'll learn about template metaprogramming.

WE'LL COVER THE FOLLOWING

Template Metaprogramming

How this all started:

- 1994 Erwin Unruh discovered template metaprogramming by accident.
- His program failed to compile but calculated the first 30 prime numbers at compile-time.
- To prove his point, he used the error messages to display the first 30 prime numbers.

Let's have a look at the screenshot of the error:

```
Type `enum{}' can't be converted to txpe `D<2>' ("primes.cpp",L2/C25).
Type `enum{}' can't be converted to txpe `D<3>' ("primes.cpp",L2/C25).
Type `enum{}' can't be converted to txpe `D<5>' ("primes.cpp",L2/C25).
Type `enum{}' can't be converted to txpe `D<7>' ("primes.cpp",L2/C25).
Type `enum{}' can't be converted to txpe `D<11>' ("primes.cpp",L2/C25).
Type `enum{}' can't be converted to txpe `D<13>' ("primes.cpp",L2/C25).
Type `enum{}' can't be converted to txpe `D<17>' ("primes.cpp",L2/C25).
Type `enum{}' can't be converted to txpe `D<19>' ("primes.cpp",L2/C25).
Type `enum{}' can't be converted to txpe `D<23>' ("primes.cpp",L2/C25).
Type `enum{}' can't be converted to txpe `D<29>' ("primes.cpp",L2/C25).
```

We have highlighted the important parts in red. We hope you can see the pattern. The program calculates at compile-time the first *30* prime numbers. This means template instantiation can be used to do math at compile-time. It gets even better. Template metaprogramming is Turing-complete and can, therefore, be used to solve any computational problem. Of course, Turing-completeness holds only in theory for template metaprogramming because the recursion depth (at least 1024 with C++11) and the length of the names which are generated during template instantiation provide some limitations.

Calculating at Compile-Time

The `Factorial` program is the *Hello World* of template metaprogramming.

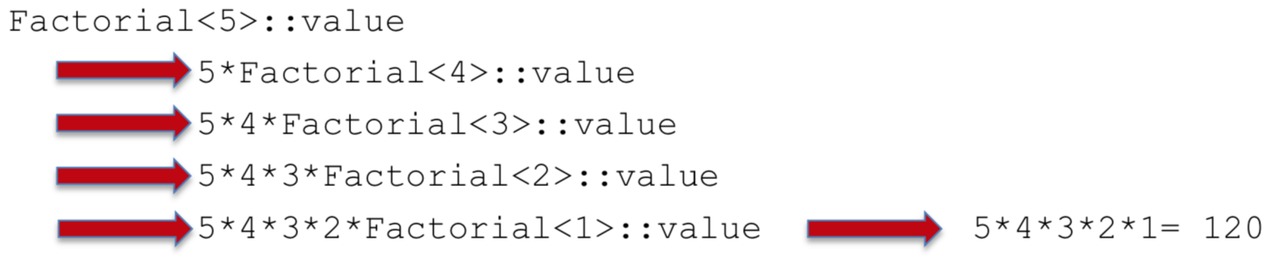
```
template <int N>
struct Factorial{
    static int const value= N * Factorial<N-1>::value;
};

template <> struct Factorial<1>{
    static int const value = 1;
};

std::cout << Factorial<5>::value << std::endl;
std::cout << 120 << std::endl;
```

The call `Factorial<5>::value` in line 10 causes the instantiation of the primary or general template in line 3. During this instantiation, `Factorial<4>::value` will be instantiated. This recursion will end if the fully specialized class template `Factorial<1>` (line 6) kicks in as the boundary condition. Maybe, you like it more pictorial.

The following picture shows this process.



Assembler Instructions

From the assemblers point of view, the `Factorial<5>::value` boils down to the constant `0x78`, which is 120.

```
mov 0x78, %esi
mov 0x601060, %edi
...
mov 0x78, %esi
mov 0x601060, %edi
...
```

Type Manipulations

Manipulating types at compile-time is typically for template metaprogramming.

```
template <typename T>
struct RemoveConst{
  typedef T type;
};

template <typename T>
struct RemoveConst<const T>{
  typedef T type;
};

int main(){
  std::is_same<int, RemoveConst<int>::type>::value;       // true
  std::is_same<int, RemoveConst<const int>::type>::value; // true
}
```

Explanation

In the code, we have defined the class template `removeConst` in two versions. We have implemented `removeConst` the way `std::remove_const` is probably implemented in the type-traits library.

`std::is_same` from the type-traits library helps us to decide at compile-time if both types are the same. In case of `removeConst<int>`, the first or general class template kicks in; in case of `removeConst<const int>`, the partial specialization for `const T` applies. The key observation is that both class templates return the underlying type in lines 3 and 8, therefore, the constness is removed.

This kind of technique, which is heavily used in the type-traits library, is a compile-time `if` on types.

To jump into more details of type traits click here.

Metadata and Metafunctions

At compile-time, we speak about metadata and metafunctions instead of data and functions.

- **Metadata:** Types and integral types that are used in metafunctions.
- **Metafunctions:** Functions that are executed at compile-time. Class templates are used to implement metafunctions.
- Return their value by `::value`.

```
template <>
struct Factorial<1>{
   static int const value = 1;
};
```

- Return their type by `::type`.

```
template <typename T>
struct RemoveConst<const T>{
   typedef T type;
};
```

Functions vs Meta Functions

From the conceptual view, it helps a lot to compare functions and metafunctions.

Characteristics	Functions	Metafunctions
Call	`power(2,10)`	`Power<2,10>::value`
Execution Time	Runtime	Compile-time
Arguments	Function arguments	Template arguments
Arguments and return value	Arbitrary values	Types, non-types and templates
Implementation	Callable	Class template
Data	Mutable	Immutable
Modification	Data can be modified	New data are created
State	Has state	Has no state

What does the table above mean for a concrete function and a concrete metafunction?

Function

```
int power(int m, int n){
    int r = 1;
    for(int k=1; k<=n; ++k){
        r *= m;
    }
    return r;
}
```

Metafunction

```
template<int m, int n>
struct Power{
   static int const value = m * Power<m, n-1>::value;
};

template<int m>
struct Power<m, 0>{
   static int const value =1;
};
```

> Function arguments go into round ( ) braces and template arguments go

> into sharp <> braces.

```
int main(){
   std::cout << power(2, 10) << std::endl;            // 1024
   std::cout << Power<2, 10>::value << std::endl;    // 1024
}
```

Pure Functional Sublanguage

- Template metaprogramming is
  - an embedded pure functional language in the imperative language C++.
  - Turing-complete. Turing-complete means, that all can be calculated what is calculatable.
  - an intellectual playground for C++ experts.
  - the foundation for many boost libraries.

> The template recursion depth is limited.
>
> - C++03: 17
> - C++11: 1024

In the next lesson, we'll look at a few examples of template metaprogramming.