https://github.com/zwimer/TemplateMetaTutorial

This presentation is **under construction**. External files above

C++ Template Meta

A basic introduction to basic C++ techniques used in template metaprogramming.

Important reminder

Otherwise identical classes with different template classes are **NOT** the same class!

For example:

- list<bool> != list<char>
- vector<int> != vector<double>
- list< list<int> > != list< list<char> >

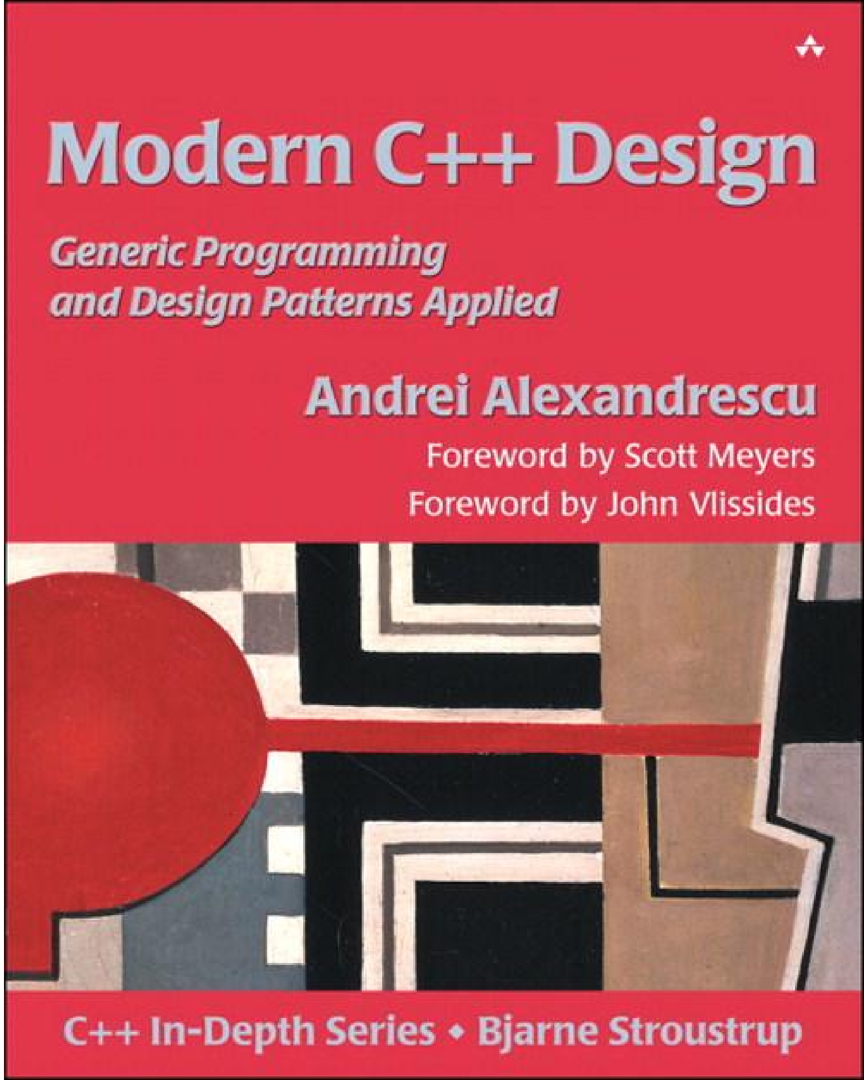
What you will learn

I will be teaching you using this book ->

We will be going over chapter's 2 and 3, techniques and typelists respectively.

If you would rather read it directly than listen to me, the links is here:

https://www.mimuw.edu.pl/~mrp/cpp/SecretCPP/Addison-Wesley%20-%20Modern%20C++%20Design.%20Generic%20Programming%20and%20Design%20Patterns%20Applied.pdf



What is TMP?

What is it?

- Code that is ‘run’ and evaluated at compile time
- ‘Compile time programming’
  - Immutable objects
  - Functional programming
- Can create
  - Data Structures
  - Compile time constants
  - Functions
- Takes advantage of templates to do this

Factorial: Run-Time vs. Compile-Time

```cpp
// Run time programming
unsigned int factorial(unsigned int n) {
        return n == 0 ? 1 : n * factorial(n - 1);
}


// Usage examples:
// factorial(0) would yield 1;
// factorial(4) would yield 24.
```

// Everything is evaluated at run-time

//Slower to run, faster to compile

```cpp
// Template Meta
//Recursive case
template <unsigned int n> struct factorial {
        enum { value = n * factorial<n - 1>::value };
};

// Base case
template <> struct factorial<0> {
        enum { value = 1 };
};

// Usage examples:
// factorial<0>::value would yield 1;
// factorial<4>::value would yield 24.
```

// Everything is evaluated at compile-time

//Faster to run, slower to compile

Factorial: Why does it work?

- <u>Remember</u>: factorial<N> is a class!
- factorial<4> != factorial<3>
    - They are <u>***different classes!***</u>
- Enum's are like static const ints
    - Though they are not the same, but that isn't relevant at the moment
- *Enums **<u>must</u>** be evaluated at compile time!*

```cpp
// Template Meta
template <unsigned int N> struct factorial {
        enum { value = N * factorial<N - 1>::value };
};

template <> struct factorial<0> {
        enum { value = 1 };
};

// Usage examples:
// factorial<0>::value would yield 1;
// factorial<4>::value would yield 24.

// Everything is evaluated at compile-time
```

Factorial: Why does it work?

- <u>Remember</u>: factorial<N> is a class!
- factorial<4> != factorial<3>
  - They are ***<u>different classes!</u>***
- Enum's are like static const ints
  - Though they are not the same, but that isn't relevant at the moment
- *Enums **<u>must</u>** be evaluated at compile time!*

Thus, the **<u>class</u>** factorial<4> has a will always have an enum 'value' with a value of N*factorial<N-1>::value that is *defined* at compile time.

```
// Template Meta
template <unsigned int N> struct factorial {
    enum { value = N * factorial<N - 1>::value };
};

template <> struct factorial<0> {
    enum { value = 1 };
};

// Usage examples:
// factorial<0>::value would yield 1;
// factorial<4>::value would yield 24.

// Everything is evaluated at compile-time
```

Weird Notation?

- Why do we have to say factorial<4>::value?
- <u>factorial<4> is a class</u>

- We want the value of the enum ‘value’ within the class factorial<4>

factorial<4>::value

```
// Template Meta
template <unsigned int N> struct factorial {
        enum { value = N * factorial<N - 1>::value };
};

template <> struct factorial<0> {
        enum { value = 1 };
};

// Usage examples:
// factorial<0>::value would yield 1;
// factorial<4>::value would yield 24.

// Everything is evaluated at compile-time
```

Did I peak your interest?

1. If you just came to see what this was on, hopefully I intrigued you enough to stay.

2. Next I am going to show you a few basic techniques

3. Before I do, <u>questions</u> on what TMP is?

Basic Techniques

Things to note

- I am going to teach you in **C++99**
- Most of what you are about to learn is *now* built into c++11, c++14, c++17, or their stl libraries.
- The rest exists in other libraries such as stl Loki, Blitz++, boost mpl, ipl, and boost::hana

Why Learn LEgacy Techniques?

1. These are fundamental techniques used in TMP
2. Many of these you will use anyway, just with a pretty wrapper around them
3. **To help you learn the TMP way of thinking, and better understand how to program in TMP**

Common Practice in TMP

1. Macros used as wrappers
   - Normally macros should be avoided, but in TMP it is common to have them wrappers
   - Functions can also be used as wrappers, *but they add overhead*
   - Not always all caps when wrappers
2. Structs with a typedef or enum that stores the result of a computation
   - Structs like this will often replace variables

```
// Template Meta
template <unsigned int N> struct _Factorial {
        enum { value = N * _Factorial<N - 1>::value };
};

template <> struct _Factorial<0> {
        enum { value = 1 };
};

// Factorial Wrapper
#define factorial(x) _Factorial<x>::value

// Usage examples:
// factorial(0) would yield 1;
// factorial(4) would yield 24.
```

Template Specialization

- One of the key elements in TMP
- When a template function / class is called, C++ algorithm matches it's call to the 'closest' matching 'most specialized' template it can

```
// Template Meta
template <unsigned int N> struct _Factorial {
	enum { value = N * _Factorial<N - 1>::value };
};

//Note that here we have template <>
//This is FULL template specialization
//We then specify what arguments in the class name
template <> struct _Factorial<0> {
	enum { value = 1 };
};

// Factorial Wrapper
#define factorial(x) _Factorial<x>::value

 // Usage examples:
// factorial(0) would yield 1;
// factorial(4) would yield 24.
```

Partial Template Specialization

- One of the key elements in TMP
- Note: This is **ONLY** allowed for classes and structs. It is not allowed for functions.

```
// Template Meta
template <int N, int N2> struct Division {
        enum { value = N / N2 };
};

//Note that here we have only one int in our template
//This is partial template specialization.
//We then specify what the arguments are below
template <int N> struct Division<N, 0> {
        enum { value = INT_MAX };
};

 // Usage examples:
// Division<4,2>::value would yield 1;
// Division<4,0>::value would yield INT_MAX.
```

Static assertions (Compile Time Assertions)

- Now standard in c++11 via static_assert

- TMP debugging is oft trying to make the program simply compile
- A simple compile time assertion
    - Makes debugging easier
    - Clearer error messages

- Trivial method without TMP.
    - Do you foresee any shortcomings?

```
//If A is false, char[0] is called, which is illegal
//If A is true, char[1] is called, then goes out of scope
#define STATIC_CHECK(A) {                                   \
        char test[ (A) ? 1 : 0 ];                           \
}

// Usage examples:
// STATIC_CHECK( 1 + 1 == 2 ) would compile
// STATIC_CHECK( 1 + 1 == 3 ) would not compile
```

Static assertions (Compile Time Assertions)

- Now standard in c++11 via static_assert

- Compiler may throw a warning instead of error
- What if you want a custom error message?

- What about a trivial TMP Method that utilizes incomplete instantiation?

```
//Declare the struct
template <bool> struct CompileTimeError;

//Define the struct for true
template <> struct CompileTimeError<true> {};

// Wrapper
#define STATIC_CHECK(A) CompileTimeError<A>()

// Usage examples:
// STATIC_CHECK( 1 + 1 == 2 ) would compile
// STATIC_CHECK( 1 + 1 == 3 ) may yield, depending
// on your compiler:
```

```
a.cpp: In function 'int main()':
a.cpp:8:29: error: invalid use of incomplete type 'struct CompileTi
meError<false>'
  CompileTimeError<2+1 == 2>();
                             ^
a.cpp:3:23: error: declaration of 'struct CompileTimeError<false>'
 template<bool> struct CompileTimeError;
                       ^
```

Static assertions (Compile Time Assertions)

- Now standard in c++11 via static_assert

- This is better, but what if we want a custom error message for <u>each</u> static assert?
  - Sidenote: Copy pasting and changing the name of the struct is bad...

```
//Declare the struct
template <bool> struct CompileTimeError;

//Define the struct for true
template <> struct CompileTimeError<true> {};

// Wrapper
#define STATIC_CHECK(A) CompileTimeError<A>()

// Usage examples:
// STATIC_CHECK( 1 + 1 == 2 ) would compile
// STATIC_CHECK( 1 + 1 == 3 ) may yield, depending
// on your compiler:
```

```
a.cpp: In function 'int main()':
a.cpp:8:29: error: invalid use of incomplete type 'struct CompileTi
meError<false>'
  CompileTimeError<2+1 == 2>();
                            ^
a.cpp:3:23: error: declaration of 'struct CompileTimeError<false>'
 template<bool> struct CompileTimeError;
                       ^
```

Static assertions (Compile Time Assertions)

- Now standard in c++11 via static_assert

UNDER CONSTRUCTION

- Macros to the rescue!
- Macros are not uncommon in TMP.
  - And smart usage can lead to better code

Mapping to Types

- Standard in many TMP libraries. Ex. Boost::Hana::Type

- One of the key elements in TMP
- Allow values and types to be declared **without** being instantiated
  - Save space
  - Save time
  - Prevent side effects
  - Allow compile time manipulation
- We **will** use these later.

```
//Map an integer to a type
template <int N> struct IntToType {
	enum { value = N };
};

//Map a type to a type
template <class T> struct TypeToType {
	typedef T value;
};
```

Second point

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua

Incididunt ut labore et dolore

Consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua

XX%

Use this slide to show a major stat. It can help enforce the presentation’s main message or argument.

Final point

A one-line description of it

“This is a super-important quote”

– From an expert

This is the most important takeaway that everyone has to remember.

THANKS!

Contact us:

Your Company
123 Your Street
Your City, ST 12345

no_reply@example.com
www.example.com