C++ Template Meta

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

Github Repo

The presentation and all of the code are online on github, with an OSI license.

github.com/zwimer/Template-Meta-Tutorial

Note: Much of this code is made using trivial techniques. They are **not** the best way to do either of these, it is just a demonstration of some of the basic techniques you will learn.

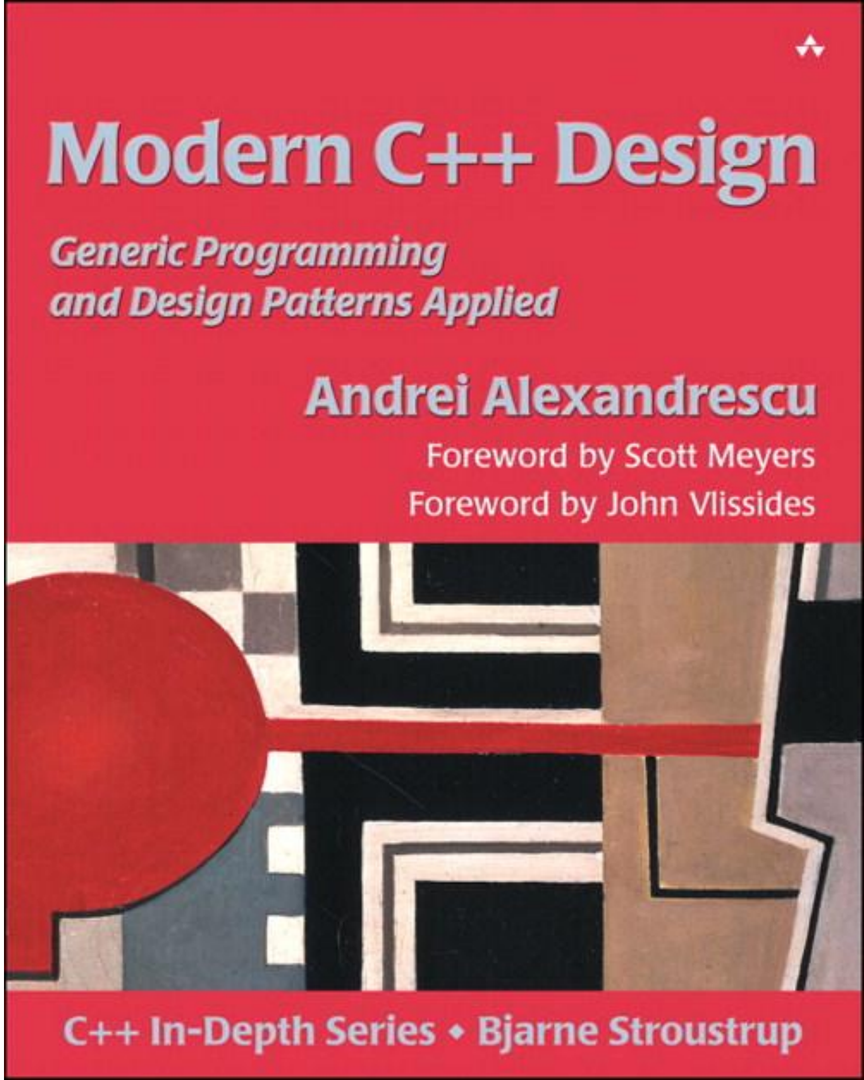
What you will learn

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



Important reminder

Otherwise identical classes with different template classes are **<u>NOT</u>** the same class!

For example:

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

What is to come:

1. What is TMP?
2. TMP: Why bother?
   a. Quick Demo
3. Ready to learn some TMP?
4. Fundamentals
5. Basics
6. The mighty Typelist
7. Out of time… but worth a mention

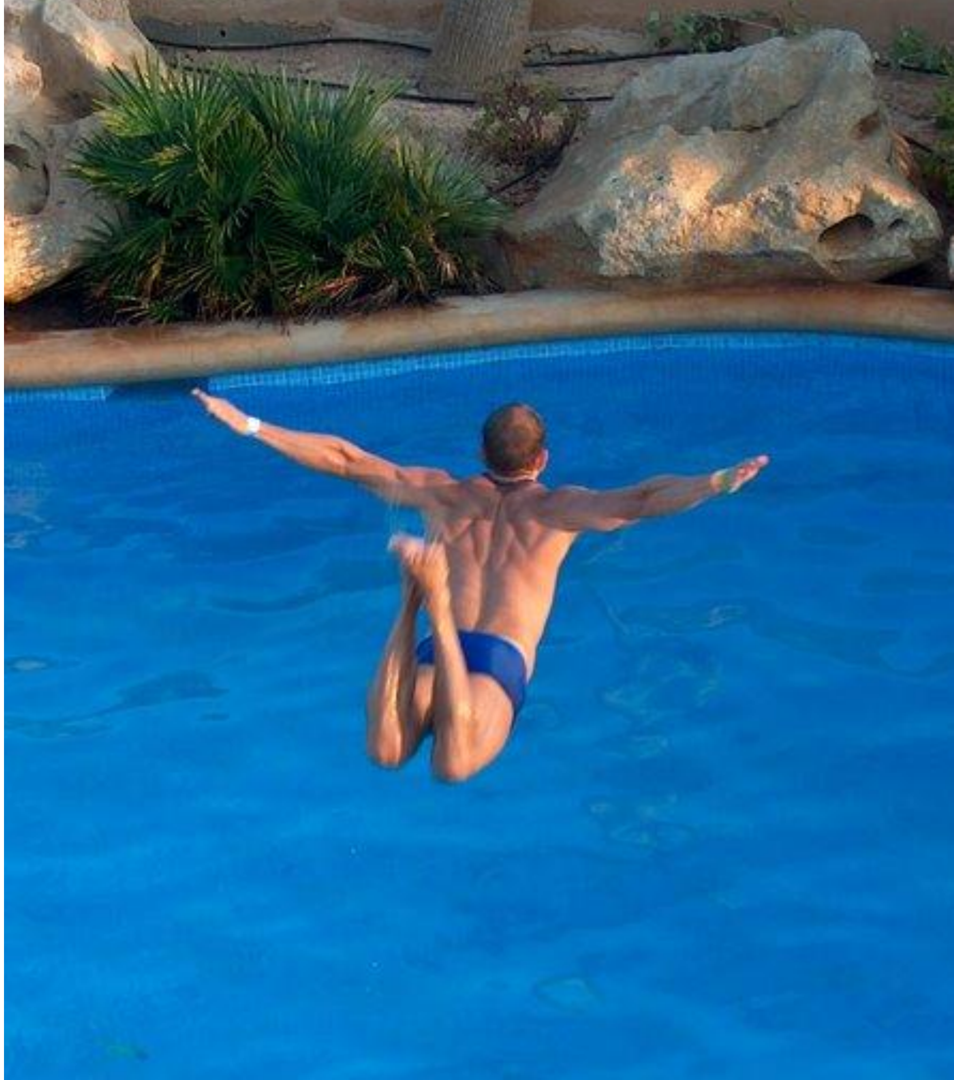
What is TMP?

What is it?

- Code that is ‘run’ and evaluated at compile time
- ‘Compile time programming’
  - Immutable objects
  - Functional programming
    - Until C++17...
- Can create
  - Data Structures
  - Compile time constants
  - Functions
- Takes advantage of **templates** to do this

Before I bore you...

- Let's just dive right into the 'hello world' of TMP.
- If you are still interested afterwards, stick around!



Factorial: Run-Time vs. Compile-Time

```
// 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

```
// 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?

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

Thus, the **class** 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.

```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
```

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, questions on what TMP fundementally is?

TMP: Why bother?

Why Use?

1. **Speed**

Why Use?

1. **<u>Speed</u>**
2. **<u>Speed</u>**

Why Use?

1. **Speed**
2. **Speed**
3. **Speed**

Why else use?

1. Modularity

Why else use?

1. Modularity
2. Robustness

Why else use?

1. Modularity
2. Robustness
3. Extensibility

Why else use?

1. Self-adjusting code during compilation
2. Abstraction *without* speed loss
3. Many run time errors become compile time errors
4. A little goes a long way
    a. Flexible adaptive code can be made

No time...

- Unfortunately we don’t have time for me to demonstrate this.
- TMP is very broad complex
    - Not a ‘learn in one sitting’ thing
- Ask me about this later if you want

Why Avoid?

1. Difficult to read
2. Harder to write
3. Different thought process required
4. Can be difficult to debug
    a. Though it could be easier too

Quick Demo

Matrix Examples

Go to github for code and explanation.

github.com/zwimer/Template-Meta-Tutorial

Select the Matrix folder

Note: This is written with C++14, and made using trivial techniques. It is **not** the best way to do either of these, it is just a demonstration

PowerSet Examples

Go to github for code and explanation.

github.com/zwimer/Template-Meta-Tutorial

Select the PowerSet folder

Note: This is written with C++14, and made using trivial techniques. It is **not** the best way to do either of these, it is just a demonstration

Ready to learn some TMP?

Things to note

- I am going to teach you in **C++98**
- 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**

Fundamentals

Common Practice in TMP

Macros used as wrappers

- Normally macros should be avoided, but in TMP it is common to have them wrappers
- *Not always all caps* when wrappers
- Macros can sometimes be used for more than just function wrappers

```cpp
// 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.
```

Common Practice in TMP

1. The auto keyword can be your best **friend**
   a. But we won’t worry about this until later

```
// 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.
```

Common Practice in TMP

1. Structs with a **typedef** or **enum** that stores the result of a computation

2. Structs like this will often replace variables *and functions*

```
// 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.
```

Common Practice in TMP

It would not be an understatement to say that **structs are the fundamental computational unit of TMP**

```cpp
// 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.
```

The magic of Sizeof

1. Returns size of the type passed in
2. Can be called on functions to get the size of the return type
3. Can be called on objects
4. **<u>DOES NOT EVALUATE THE OBJECT!</u>**
    a. Except variable length array types

The magic of Sizeof

```
// Forward declarations
class HugeClass;
HugeClass foo();


// Size of the function's return type
sizeof( foo() );
```

- *foo() is **NOT** run.*
- *sizeof(foo()) -> sizeof(HugeClass }*
- *sizeof(HugeClass) does **NOT** create a HugeClass*
- *sizeof is evaluate at compile time*
  - *Nothing is instantiated except variable length arrays*

Null Type

- A type that exists only to inform the compiler it is unimportant / a ‘null terminator’.
- Why?
  - We will get to this later, but imagine it as the 0 character at the end of a c-string

```
// An unimportant / empty type
class NullType { };
```

Classes in classes

- Classes within classes are legal in C++
- Only the items within the enclosing class can ‘see’ the enclosed class without a forward declaration

```
//A class with a function that can print out
//messages given to it by it's internal classes
class Printer {
private:
    //A class that has a function
    //that returns "Hello World!"
    class GetHi {
        private: std::string getHi() {
            return std::string("Hello World!");
        }
    };
public:
    void pntMsg() {   //Print "Hello World!"
        GetHi tmp;
        std::cout << tmp.getHi() << std::endl;
    }
};

//Main function
int main() {
    Printer p;          //Make a printer
    p.pntMsg();         //Print Hello World!
    return 0;
}
```

Basics

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

```
//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
```

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

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… We want a robust solution

```
//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

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

```
//'Catch all' constructor, can take in ANY type
template <bool> struct CompileTimeChecker {
    CompileTimeChecker(...);
};

//Specialize definition for when bool = false
//There is no (non-implicit) constructor here! Calling it illegal
template <> struct CompileTimeChecker<false> {};

// Macro Wrapper
#define STATIC_CHECK(A, msg) {                                  \
    class ERROR_##msg{};                                        \
    (void) sizeof( CompileTimeChecker<A>{ ERROR_##msg() } ); \
}

// Usage:
STATIC_CHECK( 1+2 == 3,  Math_Is_Broken) should compile
STATIC_CHECK( 1+2 != 3,  Math_Is_Broken) shouldn't compile
```

Static assertions (Compile Time Assertions)

- Now standard in c++11 via static_assert

- Quick note, in this I used initializer lists, the { } instead of ( ). <u>I did this for cleaner error messages</u>, but it is a C++11 concept.

- **To make this work for c++98, replace <u>only</u>**

  { ERROR_##msg() } with (ERROR_##msg())

```
//'Catch all' constructor, can take in ANY type
template <bool> struct CompileTimeChecker {
    CompileTimeChecker(...);
};

//Specialize definition for when bool = false
//There is no (non-implicit) constructor here! Calling it illegal
template <> struct CompileTimeChecker<false> {};

// Macro Wrapper
#define STATIC_CHECK(A, msg) {                                    \
    class ERROR_##msg{};                                          \
    (void) sizeof( CompileTimeChecker<A>{ ERROR_##msg() } ); \
}

// Usage:
STATIC_CHECK( 1+2 == 3,  Math_Is_Broken) should compile
STATIC_CHECK( 1+2 != 3,  Math_Is_Broken) shouldn't compile
```

Mapping to Types

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

- Allow values and types to be declared **without** being instantiated
    a. Prevent instantiation side effects
    b. Allow compile time manipulation
    c. Modularity
    d. Save space
    e. Save time
    f. Etc...

```
//Map an integer to a type
template <int N> struct Int2Type {
        enum { value = N };
};

//Map a type to a type
template <class T> struct Type2Type {
        typedef T value;
};
```

Type Selection

- Now standard in c++ libraries

- Akin to an if statement
  - `if (B) return T; else return U;`

```
//Map an integer to a type
template <int N> struct Int2Type {
        enum { value = N };
};

//If B is true (general case), then result = T
template <bool B, class T, class U> struct Select {
        typedef T result;
};

//If B is false, then result = U
//Since this is specialized, it takes priority
template <class T, class U> struct Select<false, T, U> {
        typedef U result;
};

// Usage:
//Select< true, Int2Type<100>, Int2Type<0> >::result::value
would yield 100
```

Type Selection

- Now standard in c++ libraries

- Akin to an if statement
  - `if (B) return T; else return U;`

Note the use of 'mapping to types'

- Allows the int to be treated as a type instead just of a constant

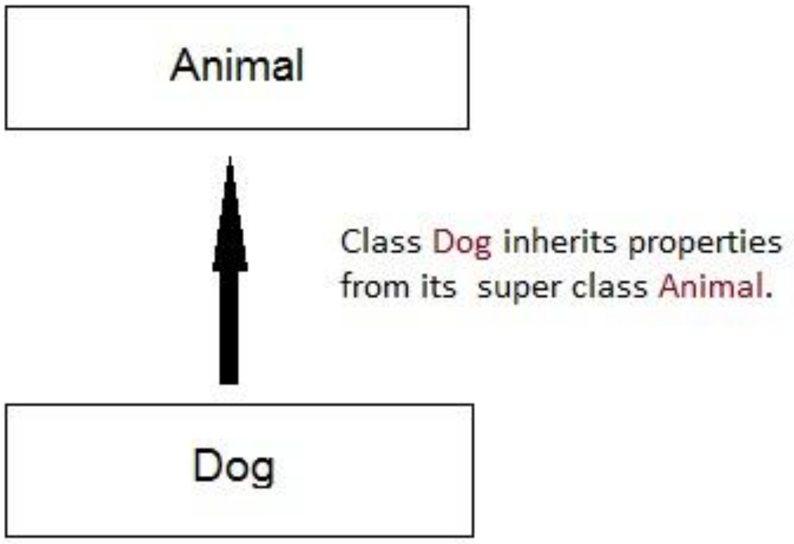
```
//Map an integer to a type
template <int N> struct Int2Type {
        enum { value = N };
};

//If B is true (general case), then result = T
template <bool B, class T, class U> struct Select {
        typedef T result;
};

//If B is false, then result = U
//Since this is specialized, it takes priority
template <class T, class U> struct Select<false, T, U> {
        typedef U result;
};

// Usage:
//Select< true, Int2Type<100>, Int2Type<0> >::result::value
would yield 100
```

Ready for a toughie?

Check Inheritance

- Now std::is_base_of

- Often times it will be important for a class to know if one class is derived from another
- For example: Curiously Recurring Template Pattern



Check Inheritance

- Now std::is_base_of

- Don't want to hardcode this by hand.
    - Bad practice
    - Not modular
    - Not robust
    - Requires upkeep
    - Could be dangerous
    - Time consuming
    - Could have extraneous code
    - Could have complex error messages
    - Etc...

- **Just a bad idea**

Quick reminder

1. The ellipsis (…) in C++ is generally used for variadic arguments / templates.
2. It has the nice effect of accepting any and all arguments
3. Also, classes can contain other classes.

```
// Function that takes an ellipsis argument
void hi(...) {
    cout << "Hi\n";
}

// Usage examples:
// factorial() would print Hi.
// factorial(5) would print Hi.
// factorial("Bye") would print Hi.
// factorial( 2, "Bye", NULL ) would print Hi.

// Every argument is always accepted
```

Check Inheritance

- Now std::is_base_of

Go to github for code and explanation.

github.com/zwimer/Template-Meta-Tutorial

Select the Inheritance folder

Note on C++11's std::is_base_of

- If both arguments are the same class, is_base_of evaluates to true

Slow down !

I know I went fast, but you need the basics before I show you the glue.

Questions?

- **Please ask!**
- These basics are some of the fundamental building blocks of template meta programming
- If You don't understand these, you won't understand what is to come

The mighty Typelist

Quick Note: More Template Specialization

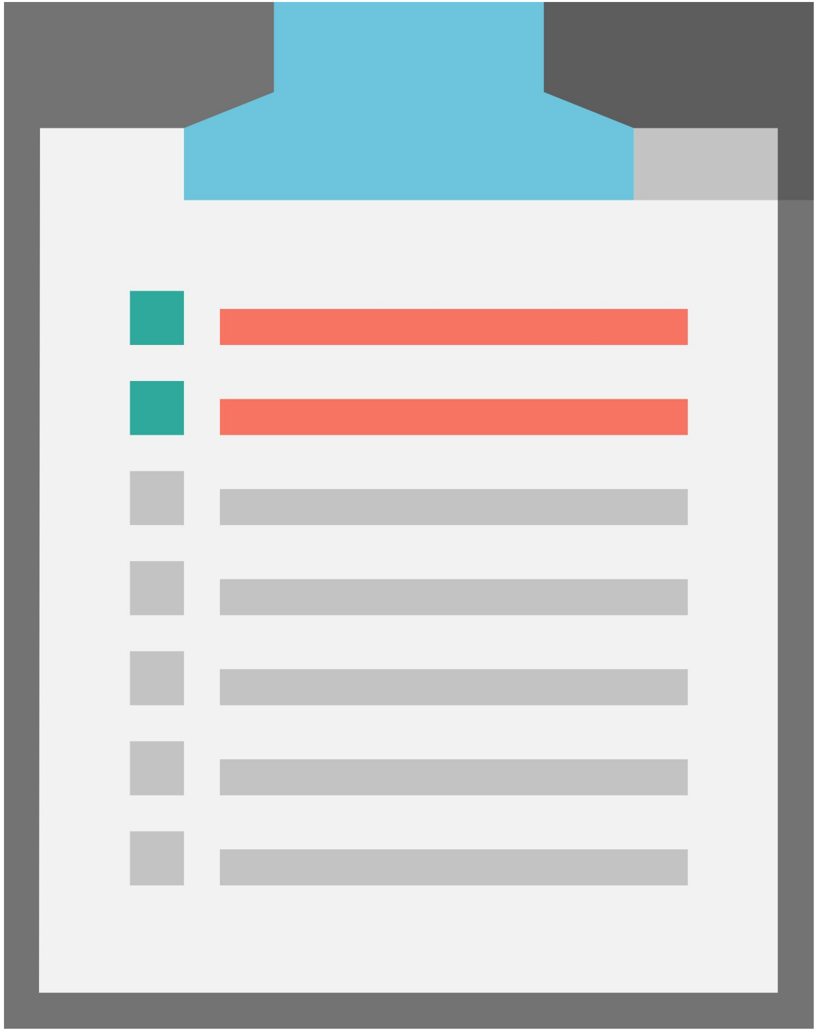
- You can template a partial template specialization that takes a templated class with the new template parameters as parameters as the template parameter
  - … I know that is a mouthful, so look right

```
//Simple Tuple class
template <class T, class U> struct Tuple {
        typedef T Head;
        typedef U Tail;
};

//Declare a CopyTuple class
template <class T> struct CopyTuple;

//Add a template to this specialization
template <> template <class T, class U>
struct CopyTuple< Tuple<T,U> > {
        typedef Tuple<T, U> result;
};

// Usage examples:
// typedef CopyTuple< A >::result B yields that is the same type
that A is
```

What is a typelist?

- Now generally made via a tuple of Type2Types

- A TL is a type whose purpose is simply to be a list of types
- Why does this matter? More later, let’s build it first!



Basic Typelist

1. Simply typedefs the two template arguments to Head and Tail
2. Can place a Typelist in another typelist to add multiple types
3. But this has many shortcomings, so <u>how can we improve this</u>?

```
//A basic Typelist
template <class T, class U> struct Typelist{
        typedef T Head;
        typedef U Tail;
};

// Usage examples:
// Typelist<char, int> is a list of a char and an integer
```

TypeList

1. Simply typedefs the two template arguments to Head and Tail
2. Can place a Typelist in the Tail of another typelist to add multiple types
3. Why is this better?
   - It lends itself better to functional programming
   - Remember, TMP often lends itself to functional programming

```
//A class that denotes the end of a TL
struct NullType;

//A simple Type List
template <class T, class U> struct Typelist{
        typedef T Head;
        typedef U Tail;
};

// Usage examples:
// Typelist<char, Typelist<int, NullType> > is a list of a
char and an integer
```

TypeList

1. Simply typedefs the two template arguments to Head and Tail
2. Can place a Typelist in the Tail of another typelist to add multiple types
3. Why is this better?
   - It lends itself better to functional programming
   - Remember, TMP often lends itself to functional programming

```
//A class that denotes the end of a TL
struct NullType;

//A simple Type List
template <class T, class U> struct Typelist{
        typedef T Head;
        typedef U Tail;
};

// Usage examples:
// Typelist<char, Typelist<int, NullType> > is a list of a
char and an integer
```

Linearizing TypeList

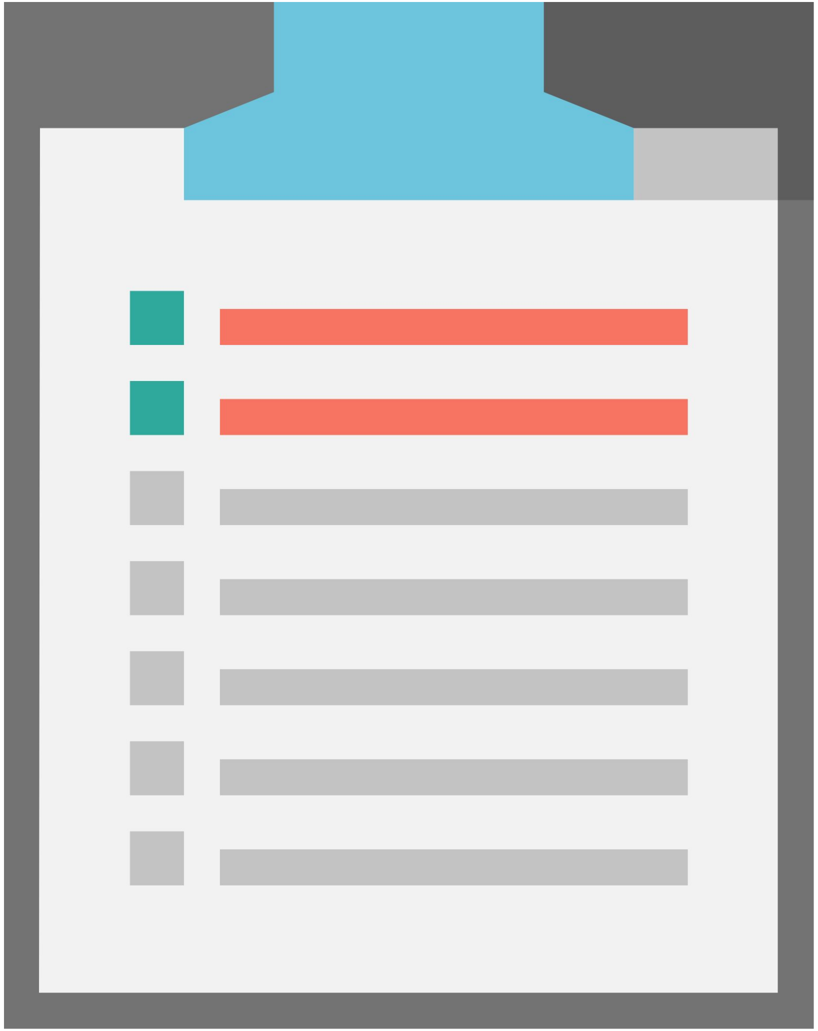
The following is for C++98. **There is a better way to do this with C++11** ! With variadic arguments, we will be able to define a single function that does the job of all of these macros! C++11 also allows for default template arguments !

```
//A class that denotes the end of a TL
struct NullType;

//A simple Type List
template <class T, class U> struct Typelist {
        typedef T Head;
        typedef U Tail;
};


// The C++98 Method ( C++11 is MUCH better for this )
#define TYPELIST_1(T1) Typelist<T1, NullType>
#define TYPELIST_2(T1, T2) Typelist<T1, TYPELIST_1(T2) >
#define TYPELIST_3(T1, T2, T3) Typelist<T1, TYPELIST_2(T2, T3) >
// ...
// ...

// In C++11, none of this is needed, you can make a struct do it for
you

//Usage Example: TYPELIST_2(char, int) makes a typelist of a char
and an int
```

A typelist is great but...

- Now generally made via a tuple of Type2Types

- What if we want to modify it?
- What can we do with it?
- Can we extend it?
- What about indexing the list?



Typelist Indexing

1. Accessing an element in a typelist like one would an array
2. Can be done with functional programming
3. Can make a macro wrapper to make it more user friendly

```
// Below assumes no errors. If you don't like this, feel free
// to throw in some of the static asserts you learned below!

//Declare the TypeAt struct
template <class T, unsigned int i> struct TypeAt;

// Get the i'th index of a typelist: Base case, i = 0.
template <> template <class T, class U>
struct TypeAt<Typelist<T, U>, 0> {
        typedef T result;
};

// Get the (i-1)'th index of the typelist U
template <> template <class T, class U, unsigned int i >
struct TypeAt<Typelist<T, U>, i> {
        typedef typename TypeAt<U, i-1>::result result;
};

//Usage: typedef TYPELIST_2(char, int) TL;
TypeAt<TL, 0>::result is a character
TypeAt<TL, 1>::result is an integer
```

TypeList Appending

What if we want to append something to a typelist? Well, let's first define what that means. Let U be the what is being appended to the typelist T. Then we can say that:

```
//A class that denotes the end of a TL
struct NullType;

//A simple Type List
template <class T, class U> struct Typelist{
        typedef T Head;
        typedef U Tail;
};

// Usage examples:
// Typelist<char, Typelist<int, NullType> > is a list of a
char and an integer
```

Typelist Appending

What if we want to append something to a typelist? Well, let's first define what that means. Let U be the what is being appended to the typelist T. Then we can say that:

1. If T is Null and U is Null, return Null

```
//A class that denotes the end of a TL
struct NullType;

//A simple Type List
template <class T, class U> struct Typelist{
        typedef T Head;
        typedef U Tail;
};

// Usage examples:
// Typelist<char, Typelist<int, NullType> > is a list of a char and an integer
```

Typelist Appending

What if we want to append something to a typelist? Well, let's first define what that means. Let U be the what is being appended to the typelist T. Then we can say that:

1. If T is Null and U is Null, return Null
2. If T is Null and U is not, return a typelist containing only U

```
//A class that denotes the end of a TL
struct NullType;

//A simple Type List
template <class T, class U> struct Typelist{
        typedef T Head;
        typedef U Tail;
};

// Usage examples:
// Typelist<char, Typelist<int, NullType> > is a list of a char and an integer
```

Typelist Appending

What if we want to append something to a typelist? Well, let's first define what that means. Let U be the what is being appended to the typelist T. Then we can say that:

1. If T is Null and U is Null, return Null
2. If T is Null and U is not, return a typelist containing only U
3. If T is Null and U is a typelist, then return U

```
//A class that denotes the end of a TL
struct NullType;

//A simple Type List
template <class T, class U> struct Typelist{
        typedef T Head;
        typedef U Tail;
};

// Usage examples:
// Typelist<char, Typelist<int, NullType> > is a list of a char and an integer
```

Typelist Appending

What if we want to append something to a typelist? Well, let's first define what that means. Let U be the what is being appended to the typelist T. Then we can say that:

1. If T is Null and U is Null, return Null
2. If T is Null and U is not, return a typelist containing only U
3. If T is Null and U is a typelist, then return U
4. If T is not null, append T::Head to Append<T::Tail, U>

```
//A class that denotes the end of a TL
struct NullType;

//A simple Type List
template <class T, class U> struct Typelist{
        typedef T Head;
        typedef U Tail;
};

// Usage examples:
// Typelist<char, Typelist<int, NullType> > is a list of a char and an integer
```

TypeList Appending

Let U be the what is being appended to the typelist T. Then we can say that:

1. If T is Null and U is Null, return Null
2. If T is Null and U is not, return a typelist containing only U
3. If T is Null and U is a typelist, then return U
4. If T is not null, append T::Head to Append<T::Tail, U>'s result

```cpp
//Declare the Append struct
template <class T, class U> struct Append;

template <> struct Append<NullType, NullType> {
        typedef NullType result;
};

template <class U> struct Append<NullType, U> {
        typedef TypeList<U, NullType> result;
};

template <> template <class T, class U>
struct Append<NullType, Typelist<T, U> {
        typedef TypeList<T, U> result;
};

template <> template <class Head, class Tail, class U>
struct Append<Typelist<Head, Tail>, U> {
        typedef Typelist<Head, typename Append<Tail, U>::result> result;
};

// Usage examples: typedef TYPELIST_2(char, bool) TL
// Append<TL, int>::result is a typelist of a character, bool, and an integer
```

Other TL operations

- We only derived two possible TL ‘functions’ due to time constraints. But are there others? Yes

1. Length
2. Indexed access
3. Search functions
4. Appending
5. Prepending
6. Inserting
7. Erasing
8. Remove duplicates
9. Replacement
10. Sorting
11. Etc.
    - It’s a list after all

Why use a TL?

1. Used all the time in TMP
2. Listing types *without* instantiation
3. Classes that require knowing types.
   a. Factories for arbitrary collections of types
4. List of classes with static functions that can be run

Why use a TL?

1. Template ‘Attribute’ design.
    a. Each template parameter represents an attribute
        i. E.g. Construction method
        ii. Equality comparison method
    b. Could store information in a TL
2. Required by other TMP design patterns
    a. Visitor patterns
3. Mixin-Based Programming in C++
4. Generating hierarchies
    a. Inheritance hierarchies

Why use a TL?

1. Type comparison
    a. Make decisions based on if a type is within a typelist
2. Create a list of functors
    a. Different hash functions for example
        i. Could use a different one depending on which type is passed in, which uses another TL
3. MultiMethods
    a. Chapter 11 if you are interested
4. It is type safe

Why use a TL?

Tuples !

- Perhaps the most useful construct in TMP
- Tuples can be implemented with TLs.

Why use a TL?

And this list goes on… and on…
and on...

No time for, but worth a mention

More Template Specialization

1. We can specify a template not only by class/value, but by traits of T too

```cpp
//General template
template <class T> struct IsPointer {
   enum { result = false };
};

//Specified template. You will notice that this
//still  takes in an argument T, but that the
//specification is simply that T is a pointer!
template <class T> struct IsPointer<T*> {
   enum { result = true };
};

// Macro Wrapper
#define isPtr(T) ((bool) IsPointer<T>::result)

// Usage: isPtr(int) should yield false
// Usage: isPtr(int*) should yield true
```

If you are interested in this, you can also to equality comparisons, modular arithmetic and more within the specification, it is worth a google.

Type Traits

- Now standardized in type_traits and hana::traits

- Together we derived isPointer
- There are dozens of other traits you can derive
- To the side are just a few of the possible traits that can be created

Table 2.1. TypeTraits<T> Members

Name	*Kind*	*Comments*
isPointer	Boolean constant	True if T is a pointer.
PointeeType	Type	Evaluates to the type to which T points, if T is a pointer. Otherwise, evaluates to NullType.
isReference	Boolean constant	True if T is a reference.
ReferencedType	Type	If T is a reference, evaluates to the type to which T refers. Otherwise, evaluates to the type T itself.
ParameterType	Type	The type that's most appropriate as a parameter of a nonmutable function. Can be either T or const T&.
isConst	Boolean constant	True if T is a const-qualified type.
NonConstType	Type	Removes the const qualifier, if any, from type T.
isVolatile	Boolean constant	True if T is a volatile-qualified type.
NonVolatileType	Type	Removes the volatile qualifier, if any, from type T.
NonQualifiedType	Type	Removes both the const and volatile qualifiers, if any, from type T.
isStdUnsignedInt	Boolean constant	True if T is one of the four unsigned integral types (unsigned char, unsigned short int, unsigned int, or unsigned long int).
isStdSignedInt	Boolean constant	True if T is one of the four signed integral types (signed char, short int, int, or long int).
isStdIntegral	Boolean constant	True if T is a standard integral type.
isStdFloat	Boolean constant	True if T is a standard floating-point type (float, double, or long double).
isStdArith	Boolean constant	True if T is a standard arithmetic type (integral or floating point).
isStdFundamental	Boolean constant	True if T is a fundamental type (arithmetic or void).

Sadly
We are out of time

If you are interested in TMP, you should look up the following

Useful to know for template meta

1. Compounded templates
2. Variadic templates
3. Template templates
4. Lambda functions
5. std::enable_if
6. if constexpr
7. constexpr
8. Tuples
9. SFINAE

<u>Libraries:</u>

1. C++ Standard Library
2. Boost::hana (I prefer over mpl)
3. Boost:mpl

Welcome to the world of Template metaprogramming: A slow descent towards utter madness

Thanks!

Contact:

Zachary Wimer

zwimer@gmail.com
zwimer.com