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> >

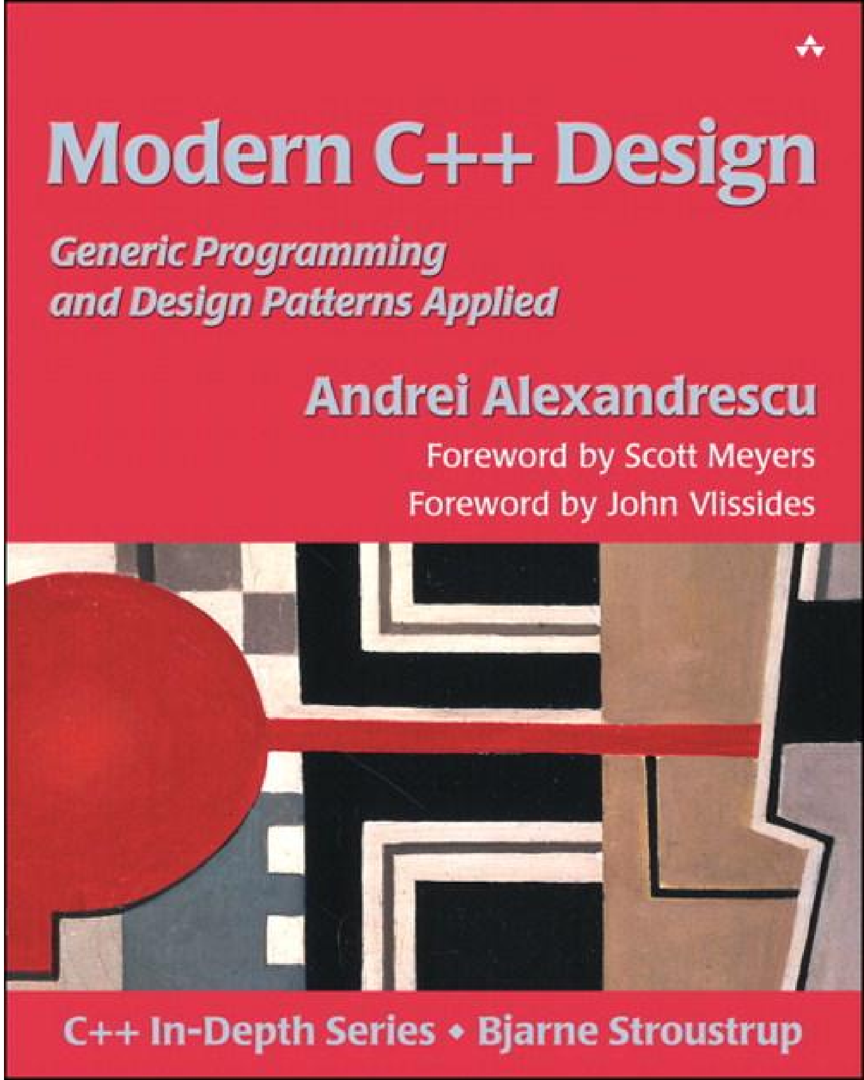
What you will learn

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



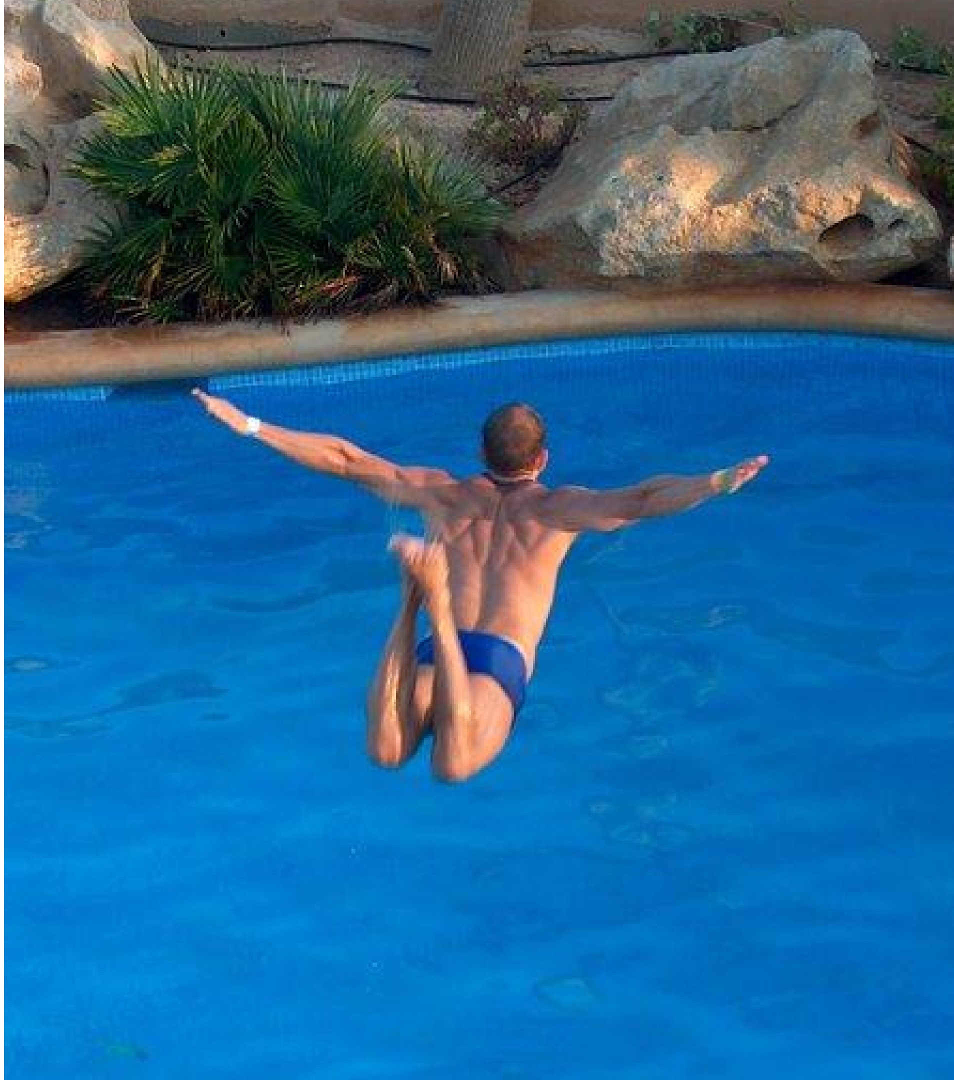
What is TMP?

What is it?

- Code that is ‘run’ and evaluated at compile time
- ‘Compile time programming’
  - Immutable objects
  - Functional programming
    - 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

```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?

- 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!*

```
// 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.

```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?
- factorial<4> is a class

- 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 fundementally is?

TMP: Should I bother?

Why Use?

1. **Speed**

Why Use?

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

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

Ready to learn TMP?

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**

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. 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

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

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

```
// 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 a compiler directive*
    - *It is evaluated at compile time*
    - *Nothing is instantiated*

Null Type

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

```
// An unimportant 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

- Macros to the rescue!
- 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++99, 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;`

```
//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
```

Note the use of ‘mapping to types’

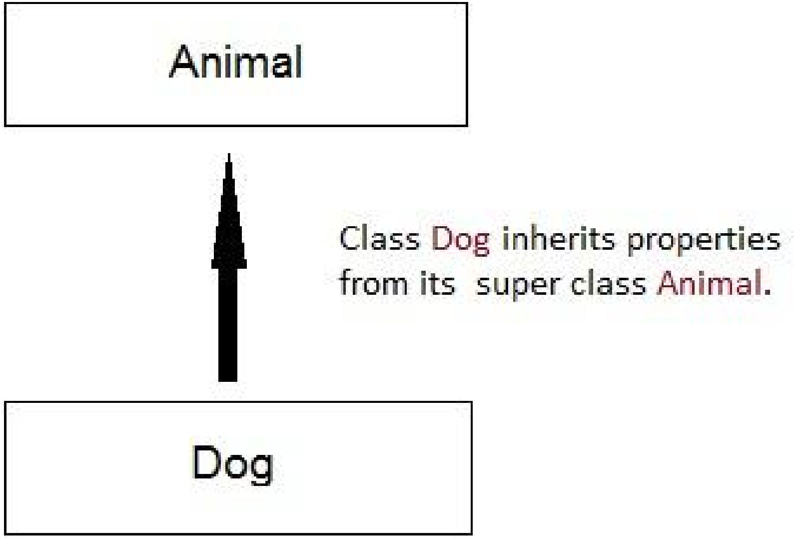
- Allows the int to be treated as a type instead just of a constant

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 <u>fundamental</u> building blocks of template meta programming

- If You don’t understand these, you won’t understand what is to come

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

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

```
//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
```

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).

The mighty Typelist

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. Variadic templates
2. Lambda functions
3. std::enable_if
4. if constexpr
5. constexpr
6. Tuples
7. SFINAE

Libraries:

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
Slack: wimerz

wimerz@rpi.edu
zwimer.com