An Introduction to Template Metaprogramming

Barney Dellar

Software Team Lead
Toshiba Medical Visualisation Systems

Caveat

- I decided to do this talk after getting thoroughly lost on the recent talk on SFINAE.
- I am ***not*** an expert on this stuff.
- I volunteered to do this talk, to give me a deadline to learn enough to not look like an idiot...
- ¡This talk contains code!

The basics

- Template Metaprogramming (TMP) arose by accident.

- Support for templates was added to C++.

- Without realising it, a Turing-complete functional language was added.

- This language is executed by the compiler. The output is C++ code.

The basics

- Because TMP was not designed, the syntax is unpleasant and unintuitive, to say the least.

- Because TMP is a functional language, the coding style is very different to standard imperative C++.

- Having said that, TMP has a reputation for being a guru-level skill, which is underserved.

- We can all learn enough to use it in our day-to-day coding.

History

- When C was developed, maths libraries implemented functions such as “square”.

- C does not support function overloading.

- So instead, *square* takes and returns doubles, and any input parameters are converted to double.

- This relies on C (and C++)’s complicated implicit numeric conversion rules...

History

- The next iteration of *square*, in C++, used function overloading.
- This gave us a version for int, a version for double, a version for float, etc etc.

```
int square(int in) {
    return in*in;
}
long double square(long double in) {
    return in*in;
}
```

Etc...

History

- The obvious next step was to allow the type for these identical functions to be generic.
- Enter templates:

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

- T acts as a wildcard in the template.

History

- Code is generated each time the template is invoked with an actual value for T.

- The actual value of T can be deduced by the compiler for function calls:

```
int i = 2;
long double d = 3.4;
auto i_squared = square(i); // int
auto d_squared = square(d); // long double
```

History

- Note that you can use “class” or “typename” when declaring a template variable.

- These are the same:

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

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

C-Style Arrays

- Another thing C++ inherited from C is that you can’t return C-style arrays from a function.

- However, if we wrap it in a struct, we can return the struct:

```
struct MyArray{
    int data_[4];
};
MyArray f(){
    MyArray a;
    return a;
}
```

C-Style Arrays

- But, what if we want to return an array with 5 elements?
- We can template the class:

```
template<typename T, int I>
struct MyArray{
   T data_[I];
};
```

- Note that the second parameter is an int, not a type.

- Template parameters can be types int (including enum, short, char, bool etc.) pointer to function, pointer to global object, pointer to data member and nullptr_t, or a type.

Hello World

The classic TMP “hello world” app is factorial calculation.

4! = 4 * 3 * 2 * 1

```
// C++.
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

```
# Haskell
factorial :: Integer -> Integer
factorial n = n * factorial (n - 1)
factorial 0 = 1
```

Hello World

```
// TMP. Executed at compile-time:
template<int N>
struct Factorial {
    static const int v = N * Factorial<N - 1>::v;
};

template<>
struct Factorial<0> {
    static const int v = 1;
};

const int f = Factorial<4>::v; // f = 24
```

Function Definition

```
template <int N>
struct Factorial {
    static const int v = N * Factorial<N-1>::v;
};
template <>
struct Factorial<0> {
    static const int v = 1;
};

int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

Input

Return Value

Pattern Matching

- The compiler has rules for which function overload to use.
- If the choice is ambiguous, the compilation will fail.
- If there is no match, the compilation will fail.
- The compiler will choose a more specific overload over a more generic one.

Pattern Matching

- The compiler will ignore overloads where the signature does not make sense.

- For example:

```
template <typename T>
typename T::value FooBar(const T& t) {
  return 0;
}

// Does not compile. int has no “value” subtype.
FooBar(0);
```

Pattern Matching

- However, if we add in a different overload that **DOES** work for int, it **WILL** compile.

```
template <typename T>
typename T::value FooBar(const T& t) {
  return 0;
}

int FooBar(int i) { return 0; }

// *Does* compile. The compiler chooses the new
//  overload, and ignores the ill-formed one.
FooBar(0);
```

Pattern Matching

- The parameter ellipsis is often used in TMP.

- It is the most generic set of arguments possible, so is useful for catching default cases in TMP:

```
template <typename T>
typename T::value FooBar(const T& t) {
    return 0;
}
int FooBar(int i) { return 0; }
void FooBar(...) {  } // Most generic case.
```

Pattern Matching

- Note that the compiler only checks the ***signature*** when deciding which overload to choose.

- If the chosen overload has a function **body** that does not compile, you will get a compilation error.

```
template <typename T>
typename T::internalType FooBar(const T& t) {
   return t.Bwahaha();
}
void FooBar(...) {  }
struct MyClass {
   using internalType  = int;
};
MyClass my_class;
FooBar(my_class); // Does not compile.
```

Type Modification

- TMP operates on C++ types as variables.

- So we should be able to take in one type, and return another...

- We use “using” (or typedef) to define new types.

Type Modification

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

template <>
struct FooToBar <Foo> {
    using type = Bar;
};

using X = FooToBar<MyType>::type; // X = MyType
using Y = FooToBar<Foo>::type;    // Y = Bar
using Z = FooToBar<Bar>::type;    // Z = Bar
```

Pointer-Removal

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

template <typename T>
struct RemovePointer <T*> {
    using type = T;
};

using X = RemovePointer<Foo>::type;  // X = Foo
using Y = RemovePointer<Foo*>::type; // Y = Foo
```

Tag Dispatch

- Tag Dispatch is a technique used to choose which code path to take at compile time, rather than at run time.

- We use tag types to make that decision.

Tag Dispatch

- Let’s look at a real example.

- Suppose we want a function that advances forward through a container using an iterator.

- If the container allows random access, we can just jump forwards.

- Otherwise we have to iteratively step forwards.

- We *could* use inheritance-based polymorphism, but that means a runtime decision.

Tag Dispatch

- Firstly, let's declare two tags:

```
struct StandardTag { };
struct RandomAccessTag { };
```

- Now, choose a tag, based on an iterator type:

```
template <typename T>
struct TagFromIterator {
    using type = StandardTag;
};
template <>
struct TagFromIterator <RandomAccessIter> {
    using type = RandomAccessTag;
};
```

Tag Dispatch

- We need two implementation functions. A standard one:

```
template <typename I>
void AdvanceImp(I& i, int n, StandardTag) {
    while (n--) {
        ++i;
    }
}
```

- And an optimised one for random access

```
template <typename I>
void AdvanceImp(I& i, int n, RandomAccessTag) {
    i += n;
}
```

Tag Dispatch

- And finally, we need a public function that hides the magic:

```
template <typename I>
void Advance(I& I, int n) {
    TagFromIterator<I>::type tag;
    AdvanceImp(i, n, tag);
}
```

Tag Dispatch

```
struct StandardTag { };
struct RandomAccessTag { };

template <typename T>
struct TagFromIterator {using type = StandardTag;};
template <>
struct TagFromIterator <RandomAccessIter> {using type = RandomAccessTag;};

template <typename I>
void AdvanceImp(I& i, int n, StandardTag) {while (n--) ++i;}
template <typename I>
void AdvanceImp(I& i, int n, RandomAccessTag) {i += n;}

template <typename I>
void Advance(I& I, int n) {
    TagFromIterator<I>::type tag;
    AdvanceImp(i, n, tag);
}
```

SFINAE

- SFINAE stands for “Substitution Failure Is Not An Error”.

- It was introduced to prevent random library header files causing compilation problems.

- It has since been adopted/abused as a standard TMP technique.

SFINAE

- SFINAE techniques rely on the compiler only choosing valid function overloads.
- You can make certain types give invalid expressions, and thus make the compiler ignore these overloads.

SFINAE

- We can use this technique to detect the presence or absence of something from a type.
- Let’s try and detect if a type exposes the operator().
- This uses the “classic” TMP sizeof trick.

SFINAE

```
template<typename T>
class TypeIsCallable{

    using yes = char(&)[1]; // Reference to an array
    using no = char(&)[2];  // Reference to an array

    template<typename C> // Detect Operator()
    static yes test(decltype(&C::operator()));

    template<typename C> // Worst match
    static no test(...);

public:

    static const bool value = sizeof(test<T>(nullptr)) == sizeof(yes);
};
```

SFINAE

- The “sizeof” trick/hack is redundant in modern C++.

- Instead, we can use constexpr, decltype and declval to reflect on types.

SFINAE

```
template <class T> class TypeIsCallable
{
    // We test if the type has operator() using decltype and declval.
    template <typename C> static constexpr
 decltype(std::declval<C>().operator(), bool) test(int /* unused */) {
        // We can return values, thanks to constexpr instead of playing with sizeof.
        return true;
    }

    template <typename C> static constexpr bool test(...) {
        return false;
    }

public:

    // int is used to give the precedence!
    static constexpr bool value = test<T>(int());
};
```

SFINAE

- We can now switch on whether an object is callable or not.
- And that switch will be decided at compile-time.
- There will be no branching in the compiled executable.

std::enable_if

- The Standard Library provides support for choosing between overloads with std::enable_if.

```
template <typename T>
typename std::enable_if<TypeIsCallable<T>::value, T>::type VerifyIsCallable(T t) {
    return t;
}
template <typename T>
typename std::enable_if<!TypeIsCallable<T>::value, T>::type VerifyIsCallable(T t)
{
    static_assert(false, "T is not a callable type");
}
VerifyIsCallable(7); // "T is not a callable type"
```

Variadic Templates

- With C++11, we can now create templates with a variable number of arguments.

- This allows us to write a function that takes a variable number of parameters in a type-safe manner.

- The syntax is odd if you’re not used to it!

Variadic Templates

- Let's look at a simple example. We will add all of the inputs to a function.
- Because we are generating code at compile time, we cannot update state, so we have to use recursion.

```
template<typename T>
T Adder(T v) {
    return v;
}

template<typename T, typename... Args>
T Adder(T first, Args... args) {
    return first + Adder(args...);
}

auto int_sum = Adder(2, 3, 4);
auto string_sum = Adder(std::string("x"), std::string("y"));
auto wont_compile = Adder(3, std::string("y"));
auto wont_compile = Adder(my_obj_1, my_obj_2);
```

Variadic Templates

- "typename... Args" Is called a *template parameter pack.*
- "Args... args" is called a *function parameter pack*.
- The general adder is defined by peeling off one argument at a time from the template parameter pack into type *T* (and accordingly, argument *first*).
- So with each call, the parameter pack gets shorter by one parameter. Eventually, the base case is encountered.

```
template<typename T, typename... Args>
T Adder(T first, Args... args) {
    return first + Adder(args...);
}
```

sizeof...

- A new operator came in with C++11.

- It's called "sizeof..." and it returns the number of elements in a parameter pack.

```
template<typename... Args>
struct VariadicTemplate{
   static int size = sizeof...(Args);
};
```

Variadic Templates: Tuple

- Now that we can construct templates with a variable number of types, we can create a “tuple” class.

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

template < typename T, typename... Ts>
struct Tuple<T, Ts...> : Tuple<Ts...>
{
   Tuple(T t, Ts... ts) :
      Tuple<Ts...>(ts...),
      head(t)
   {
   }

   T head;
};
```

Variadic Templates: Tuple

- In our tuple, an instance of a tuple-type knows about its value, and inherits from a type that knows about the next one, up until we get to a type that knows about the last one.

- So, how do we access an element in the tuple?

- Recursion!

Variadic Templates: Tuple

- Firstly, we need a helper templated on the recursion index:

```
template<int Index, typename T>
struct GetHelper;

template<typename T, typename... Ts>
struct GetHelper<0, Tuple <T, Ts...> >
{
    // select first element
    using type = T;
    using tuple_type = Tuple<T, Ts...>;
};

template<int Index, typename T, typename... Ts>
struct GetHelper<Index, Tuple <T, Ts...> > :
    public GetHelper<Index - 1, Tuple <Ts...> >
{
    // recursive GetHelper definition
};
```

Variadic Templates: Tuple

- Now we can write our Get function:

```
template<int Index, typename ...Ts>
typename GetHelper<Index, Tuple<Ts...> >::type Get(
    Tuple <Ts...> tuple
) {
    using tuple_type = GetHelper<Index, Tuple <Ts...> >::tuple_type;
    return (static_cast<tuple_type>(tuple)).head;
}

auto tuple = Tuple <int, float, string>(1, 2.3f, "help");

int i = Get<0>(tuple);    // 1
float f = Get<1>(tuple);  // 2.3
string s = Get<2>(tuple); // "help"

auto wont_compile = Get<3>(tuple);
```

Concepts

- We talked earlier about templates needing certain constraints in order to work.

- For example, the Adder function will only work if the type supports the “+” operator.

- We can constrain the template construction using tricks like std::enable_if.

- But the syntax is complicated, and the logic can be tortuous.

- Concepts is designed to simplify this.

Concepts

- In Haskell, you can limit a function so it will only compile with types of a certain “type class”.

- So a function that adds will only compile with types that implement the “Adding” type class.

- With Concepts in C++, the constraint will be optional.

Concepts

- With Concepts, you will be able to say: “This template is only valid for types with Operator +”.
- Or maybe you only want the template to work with types that have a function “void Bwahaha()”.
- The exact specification is still being worked on, but compilers will hopefully support something soon.

Closing Thoughts

- The syntax is unpleasant.
- The functional coding style is challenging if you’re used to imperative programming.
- But TMP does give us a powerful set of tools for writing generic code.

Credits

- I’ve drawn on a lot of blog posts to make this talk:

http://www.gotw.ca/publications/mxc++-item-4.htm

https://erdani.com/publications/traits.html

http://blog.aaronballman.com/2011/11/a-simple-introduction-to-type-traits/

http://accu.org/index.php/journals/442

http://oopscenities.net/2012/06/02/c11-enable_if/

http://eli.thegreenplace.net/2011/04/22/c-template-syntax-patterns

http://www.bfilipek.com/2016/02/notes-on-c-sfinae.html

http://jguegant.github.io/blogs/tech/sfinae-introduction.html

http://metaporky.blogspot.co.uk/2014/07/introduction-to-c-metaprogramming-part-1.html

Thanks

☺️