Introduction to template metaprogramming

Aleksandra Rylund Glesaaen
aleksandra@glesaaen.com

SA²C Seminar Series
February 23rd 2018

Talk outline

1. The basics, metaprogramming vocabulary
2. Tools and libraries
3. Expression templates

The basics

Vocabulary: metadata

A constant "value" accessible by calling ::value

```
struct a_value
{
  constexpr int value = ...;
};

a_value::value;
```

Made easier with C++14 template variables

(we will not use these here though, for a reason)

Vocabulary: metadata

A constant "value" accessible by calling ::value

```
struct a_value
{
  using type = a_value;
  constexpr int value = ...;
};

a_value::value;
```

Usually self referential

Vocabulary: metafunction

A "function" that takes its arguments through templates and stores the result in ::type

```
struct a_function
{
  template <int x, int y>
  using type = ...;
};

a_function<5, 6>::type;
```

Vocabulary: metafunction class

A function object that can itself be treated as a type.
Function call accessed by a nested metafunction named ::apply

```
struct a_metafunction_class
{
  template <typename Arg1, typename Arg2>
  struct apply
  {
    ...
  };
};
```

Example: Simple algebra

```
template <int N>
struct integer
{
  using type = integer<N>;
  static constexpr int value = N;
};

template <typename Arg1, typename Arg2>
struct multiply
{
  using type =
    integer<Arg1::value * Arg2::value>;
};

using five = integer<5>;
using m_nine = integer<-9>;

constexpr auto result =
  multiply<five, m_nine>::type::value;
```

Example: Simple algebra

```
template <int N>
struct integer
{
  using type = integer<N>;
  static constexpr int value = N;          metadata
};

template <typename Arg1, typename Arg2>
struct multiply
{
  using type =                             metafunction
    integer<Arg1::value * Arg2::value>;
};

using five = integer<5>;
using m_nine = integer<-9>;

constexpr auto result =
  multiply<five, m_nine>::type::value;
```

Example: Simple algebra

```
template <int N>
struct integer {...};

template <typename Arg1, typename Arg2>
struct multiply
  : integer< Arg1::value * Arg2::value > {};

struct square_f
{
  template <typename Arg>
  struct apply
    : multiply<Arg, Arg>
  {};
};

square_f::apply<five>::value;
```

Example: Simple algebra

```
template <int N>
struct integer {...};

template <typename Arg1, typename Arg2>
struct multiply
  : integer< Arg1::value * Arg2::value > {};

struct square_f
{
  template <typename Arg>
  struct apply
    : multiply<Arg, Arg>
  {};
};

square_f::apply<five>::value;
```

metafunction forwarding

metafunction class

Example: Functors

Template metaprogramming has no variables and is thus inherently a functional programming language.

Let us implement something functional, like nest:

nest(f, x, 5) = f(f(f(f(f(x)))))

Example: Nest function

```
template <typename F, typename X, unsigned N>
struct nest
    : nest<F,
           typename F::template apply<X>::type,
           N - 1> {};

template <typename F, typename X>
struct nest<F, X, 0>
  : X {};

using five = integer<5>;
nest<squared_f, five, 3>::type::value;
```

Example: Nest function

```
template <typename F, typename X, unsigned N>
struct nest
    : nest<F,
           typename F::template apply<X>::type,
           N - 1> {};
```

recursion

```
template <typename F, typename X>
struct nest<F, X, 0>
  : X {};
```

end of recursion

```
using five = integer<5>;
nest<squared_f, five, 3>::type::value;
```

$= ((5^2)^2)^2$

Example: Nest function

```
int main()
{
  using five = integer<5>;
  using result
    = nest<square_f, five, 3>::type;

  return result::value;
}
```

Example: Nest function

```
int main()
{
  using five = integer<5>;
  using result
    = nest<square_f, five, 3>::type;

  return result::value;
}
```

Compile, -O0

```
main:
  push rbp
  mov rbp, rsp
  mov eax, 390625
  pop rbp
  ret
```

Example: Nest function

```
int main()
{
  using five = integer<5>;
  using result
    = nest<square_f, five, 3>::type;

  return result::value;
}
```

Compile, -O0

```
main:
  push rbp
  mov rbp, rsp
  mov eax, 390625
  pop rbp
  ret
```

Compile, -O1

```
main:
  mov eax, 390625
  ret
```

Tools and libraries

brigand

brigand is a light-weight C++11 reimagining of boost::mpl, a metaprogramming library

brigand

brigand is a light-weight C++11 reimagining of boost::mpl, a metaprogramming library

Features:

- Functional programming utilities
- Sequence storage
- Sequence algorithms
- Arithmetic operations
- ...and more

Simplifying nest with brigand

```
#include <brigand/brigand.hpp>
using brigand::_1;

template <typename F, typename X, unsigned N>
struct nest
    : nest<F,
           typename brigand::apply<F, X>::type,
           N - 1>
{};

int main()
{
  using five = integer<5>;
  return nest<multiply<_1, _1>, five, 3>::value;
}
```

Simplifying nest with brigand

```
#include <brigand/brigand.hpp>
using brigand::_1;

template <typename F, typename X, unsigned N>
struct nest
    : nest<F,
           typename brigand::apply<F, X>::type,
           N - 1>
{};

int main()
{
  using five = integer<5>;
  return nest<multiply<_1, _1>, five, 3>::value;
}
```

Sequences at compile time

```
#include <brigand/brigand.hpp>
using brigand::_1;
using brigand::_2;

using type_list =
    brigand::list<double, short, char, int>;

using sorted_list = brigand::sort<
    type_list,
    brigand::less<brigand::sizeof_<_1>,
                  brigand::sizeof_<_2>>>;
```

metal

Provides much the same functionality as brigand

metal

Provides much the same functionality as brigand

Features:

- Lambda calculus
- Sequence storage
- Sequence transformations
- Typemaps
- ...and more

Simplifying nest with metal

```cpp
#include <metal/metal.hpp>
using metal::_1;

template <typename F, typename X, unsigned N>
struct nest
    : nest<F,
           typename metal::invoke<F, X>::type,
           N - 1>
{};

int main()
{
  using five = integer<5>;
  return nest<
      metal::bind<metal::lazy<multiply>, _1, _1>,
      five, 3>::value;
}
```

constexpr

A directive for telling the compiler that the expression is computable at compile time

C++11/C++14:

- constexpr values
- constexpr functions

C++17:

- constexpr ifs
- constexpr lambdas

nest with constexpr C++14

```
constexpr long square(long x) { return x * x; }

template <typename Function>
constexpr auto nest(Function &&f, long val,
                    long N)
{
  if (N == 0)
    return val;
  else
    return nest(std::forward<Function>(f),
                f(val),
                N - 1);
}

static_assert(nest(square, 5, 3) == 390625, "");
```

nest with constexpr C++17

```
template <typename Function>
constexpr auto nest(Function &&f, long val,
                    long N)
{
  if (N == 0)
    return val;
  else
    return nest(std::forward<Function>(f),
                f(val),
                N - 1);
}

static_assert(
  nest([](auto x) { return x * x; }, 5, 3)
    == 390625);
```

What is constexpr

- Can only call other constexpr values
- … or constexpr functions
- … or constexpr classes
  - these need a constexpr constructor

constexpr if

```
void only_int(int) {}

template <typename T>
void will_fail(T x)
{
  if (std::is_same_v<T, int>)
    only_int(x);
}

int main()
{
  will_fail(std::string{});
}
```

Compile error

constexpr if

```
void only_int(int) {}

template <typename T>
void wont_fail(T x)
{
  if constexpr(std::is_same_v<T, int>)
    only_int(x);
}

int main()
{
  wont_fail(std::string{});
}
```

Compiles

constexpr if

```
void only_int(int) {}

template <typename T>
void wont_fail(T x)
{
  if constexpr(std::is_same_v<T, int>)
    only_int(x);
}

int main()
{
  wont_fail(std::string{});
}
```

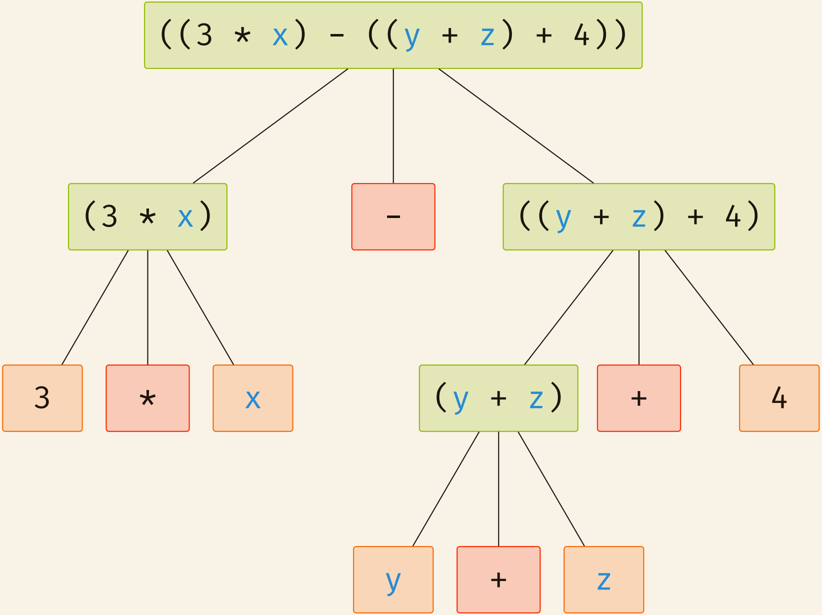
falsey branch not compiled

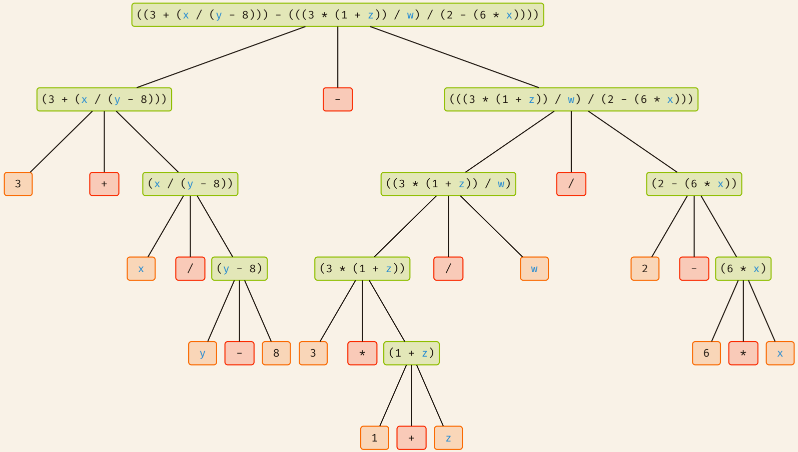
Compiles

Expression templates:
a first look

The AST (Abstract Syntax Tree)

Similar to creating lazy evaluation expressions we want to create an AST, then hook into it and determine what it does when evaluated


((3 * x) - ((y + z) + 4))
(3 * x)
-
((y + z) + 4)
3
*
x
(y + z)
+
4
y
+
z


((3 + (x / (y - 8))) - (((3 * (1 + z)) / w) / (2 - (6 * x))))
(3 + (x / (y - 8)))
-
(((3 * (1 + z)) / w) / (2 - (6 * x)))
3
+
(x / (y - 8))
((3 * (1 + z)) / w)
/
(2 - (6 * x))
x
/
(y - 8)
(3 * (1 + z))
/
w
2
-
(6 * x)
y
-
8
3
*
(1 + z)
6
*
x
1
+
z

```
ofs << "\\"
    << (3_v + "x"_v / ("y"_v - 8_v) -
        (3_v * (1_v + "z"_v) / "w"_v) /
            (2_v - 6_v * "x"_v))
           .to_tex()
    << ";\n";
```

First attempt

```
template <typename Arg1, typename Arg2>
struct plus_expr
  : integer<Arg1::value + Arg2::value>
{};

template <long N, long M>
auto operator+(integer<N>, integer<M>)
{
  return plus_expr<integer<N>, integer<M>>{};
}

int main()
{
  auto expr = five{}+eight{};

  std::cout << expr.value << std::endl;
}
```

Some obstacles

It doesn’t do complicated expressions:

```
five{} + eight{} + five{};
```

Some obstacles

It doesn't do complicated expressions:

```
five{} + eight{} + five{};
```

Possible solution:

```
template <long N, typename Arg1, typename Arg2>
auto operator+(integer<N>, plus_expr<Arg1, Arg2>)
{
  return plus_expr<integer<N>, plus_expr<Arg1, Arg2>>{};
}

template <long N, typename Arg1, typename Arg2>
auto operator+(plus_expr<Arg1, Arg2>, integer<N>)
{
  return plus_expr<plus_expr<Arg1, Arg2>, integer<N>>{};
}
```

Some obstacles

It doesn't do complicated expressions:

```
five{} + eight{} + five{}
```

Possible solution:

```
template <long N, typename Arg1, typename Arg2>
auto operator+(integer<N>, plus_expr<Arg1, Arg2>)
{
  return plus_expr<integer<N>, plus_expr<Arg1, Arg2>>{};
}

template <long N, typename Arg1, typename Arg2>
auto operator+(plus_expr<Arg1, Arg2>, integer<N>)
{
  return plus_expr<plus_expr<Arg1, Arg2>, integer<N>>{};
}
```

Tedious

Some obstacles

It doesn’t do complicated expressions:

```
five{} + eight{} + five{};
```

Possible solution:

```
template <typename Arg1, typename Arg2>
auto operator+(Arg1, Arg2)
{
  return plus_expr<Arg1, Arg2>{};
}
```

Some obstacles

It doesn’t do complicated expressions:

```
five{} + eight{} + five{};
```

Possible solution:

```
template <typename Arg1, typename Arg2>
auto operator+(Arg1, Arg2)
{
  return plus_expr<Arg1, Arg2>{};
}
```

Matches everything

Some obstacles

No obvious way to extend to new operators and types

```
eight{} - five{} / eight{};
```

”Possible solution”:

```
template <typename Arg1, typename Arg2, typename Arg3, typename Arg4>
auto operator+(plus_expr<Arg1, Arg2>, minus_expr<Arg3, Arg4>);

template <typename Arg1, typename Arg2, typename Arg3, typename Arg4>
auto operator/(minus_expr<Arg1, Arg2>, times_expr<Arg3, Arg4>);

template <typename Arg1, typename Arg2, typename Arg3, typename Arg4>
auto operator*(divide_expr<Arg1, Arg2>, plus_expr<Arg3, Arg4>);
```

Number of combinations $\sim$ (number of operations)^3

The time has finally come

The Curiously Recurring Template Pattern

The CRTP

```
template <typename T>
class Base
{};
```

```
class Derived : public Base<Derived>
{};
```

The CRTP

```
template <typename T>
class Base
{
  void base_call()
  {
    static_cast<T&>(*this).derived_call();
  }
};

class Derived : public Base<Derived>
{
  void derived_call();
};
```

The CRTP

```
template <typename T>
class Base
{
  void base_call()
  {
    static_cast<T&>(*this).derived_call();
  }
};
```

static_cast: no vtable lookup

```
class Derived : public Base<Derived>
{
  void derived_call();
};
```

Second attempt, pure types

```
template <typename T>
struct base_expr
{ };

template <long N>
struct integer : base_expr<integer<N>>
{ ... };

template <typename Arg1, typename Arg2>
struct plus_expr
    : base_expr<plus_expr<Arg1, Arg2>>
{
  using type =
    integer<Arg1::value + Arg2::value>;

  static constexpr long value = type::value;
};
```

Second attempt, pure types

```
template <typename Arg1, typename Arg2>
constexpr auto operator+(base_expr<Arg1>,
                         base_expr<Arg2>)
    -> plus_expr<Arg1, Arg2>
{
  return {};
}

static_assert(decltype(five{} + eight{} +
                       five{})::value == 18);
```

Second attempt, pure types

```
template <typename Arg1, typename Arg2>
constexpr auto operator+(base_expr<Arg1>,
                         base_expr<Arg2>)
    -> plus_expr<Arg1, Arg2>
{
  return {};
}

static_assert(decltype(five{} + eight{} +
                       five{})::value == 18);
```

limits pattern matching

Second attempt, pure types

```
template <typename Arg1, typename Arg2>
constexpr auto operator+(base_expr<Arg1>,
                         base_expr<Arg2>)
    -> plus_expr<Arg1, Arg2>
{
  return {};
}

static_assert(decltype(five{} + eight{} +
                       five{})::value == 18);
```

limits pattern matching

Doesn't actually use any of the CRTP functionality
base_expr is just a tag

Second attempt, using the CRTP

```
template <typename T>
struct base_expr
{
  constexpr auto value() const
  {
    return static_cast<T const&>(*this).value();
  }
};

template <long N>
struct integer : base_expr<integer<N>>
{
  static constexpr long value()
  {
    return N;
  }
};
```

Second attempt, using the CRTP

```
template <typename Arg1, typename Arg2>
struct plus_expr
    : base_expr<plus_expr<Arg1, Arg2>>
{
  constexpr plus_expr(base_expr<Arg1> const &le,
                      base_expr<Arg2> const &re)
      : le_{le}, re_{re} {};

  constexpr auto value() const
  {
    return le_.value() + re_.value();
  }

private:
  base_expr<Arg1> const &le_;
  base_expr<Arg2> const &re_;
};
```

Second attempt, using the CRTP

```
template <typename Arg1, typename Arg2>
constexpr auto
operator+(base_expr<Arg1> const &le,
          base_expr<Arg2> const &re)
{
  return plus_expr<Arg1, Arg2>(le, re);
}

static_assert(
    (five{} + eight{} + five{}).value() == 18);
```

Second attempt, using the CRTP

```
template <typename Arg1, typename Arg2>
constexpr auto
operator+(base_expr<Arg1> const &le,
          base_expr<Arg2> const &re)
{
  return plus_expr<Arg1, Arg2>(le, re);
}

static_assert(
    (five{} + eight{} + five{}).value() == 18);
```

Simple :)

Resources

[1] Jonathan Boccara.
Fluent {C++}.
https://www.fluentcpp.com/.

[2] Ben Deane and Jason Turner.
constexpr all the things.
CppCon 2017.

[3] Joel Falcou.
Expression templates - past, present, future.
CppCon 2015.

[4] Arne Mertz.
Simplify c++.
https://arne-mertz.de.

Thanks!

Irubataru/talks