C++ Template Metaprogramming considered sexy

Florian Sowade

August 20, 2011

About Me

- Florian Sowade
- currently studying computer science in Dortmund
- using C++ since about four years
- `@riodereelfte` on twitter
- `http://www.r9e.de`

Outline

Metaprogramming

- "programming programs"

Metaprogramming

- "programming programs"
- write code in a *metalanguage* to modify code in an *object language*

Metaprogramming

- “programming programs”
- write code in a *metalanguage* to modify code in an *object language*
- in TMP C++ is both meta and object language

Metaprogramming - Examples

- C preprocessor

Metaprogramming - Examples

- C preprocessor
- parser generators

Metaprogramming - Examples

- C preprocessor
- parser generators
- manipulating a running program using reflection APIs

Metaprogramming - Examples

- C preprocessor
- parser generators
- manipulating a running program using reflection APIs
- Macros in Lisp

Metaprogramming - C Preprocessor

```
int main() {
    int seconds = 3;

#ifdef DEBUG
    std::cout << "Sleeping for "<< seconds
              << " seconds" << std::endl;
#endif

#if defined(_WIN32)
    Sleep(seconds * 1000);
#elif defined(__unix__)
    sleep(seconds);
#else
#   error "unknown operating system"
#endif
}
```

Outline

C++

- developed by Bjarne Stroustrup since the 70s
- standardized by the ISO in 1998

C++

- developed by Bjarne Stroustrup since the 70s
- standardized by the ISO in 1998
- compiled

C++

- developed by Bjarne Stroustrup since the 70s
- standardized by the ISO in 1998
- compiled
- statically typed

C++

- developed by Bjarne Stroustrup since the 70s
- standardized by the ISO in 1998
- compiled
- statically typed
- multi-paradigm

C++

- developed by Bjarne Stroustrup since the 70s
- standardized by the ISO in 1998
- compiled
- statically typed
- multi-paradigm
  - procedural
  - object-oriented
  - generic

C++

- developed by Bjarne Stroustrup since the 70s
- standardized by the ISO in 1998
- compiled
- statically typed
- multi-paradigm
  - procedural
  - object-oriented
  - generic

C++

- developed by Bjarne Stroustrup since the 70s
- standardized by the ISO in 1998
- compiled
- statically typed
- multi-paradigm
  - procedural
  - object-oriented
  - generic
- Boost: A set of high quality C++ libraries. Many of them utilise or provide TMP

Outline

Templates - Template Function `max`

```
int main() {
    std::cout << max(23, 42) << std::endl
              << max('c', '+') << std::endl;
}
```

Templates - Template Function `max`

```
template <typename T>
    T max(T a, T b)
{
    return (a > b)
        ? a : b;
}



int main() {
    std::cout << max(23, 42) << std::endl
              << max('c', '+') << std::endl;
}
```

Templates - Template Function max

```
template <typename T>
    T max(T a, T b)
{
    return (a > b)
        ? a : b;
}
```

```
int max(int a, int b) {
    return (a > b)
        ? a : b;
}

char max(char a, char b) {
    return (a > b)
        ? a : b;
}
```

```
int main() {
    std::cout << max(23, 42) << std::endl
              << max('c', '+') << std::endl;
}
```

Templates - Overview

- classes and functions can have template parameters

Templates - Overview

- classes and functions can have template parameters
- template parameters can be types or constants

Templates - Overview

- classes and functions can have template parameters
- template parameters can be types or constants
- a template by itself yields no code
- code is generated when template is *instantiated* with parameters

Templates - Overview

- classes and functions can have template parameters
- template parameters can be types or constants
- a template by itself yields no code
- code is generated when template is *instantiated* with parameters
- template instantiation is done at compile time

Templates - Template Class Array

```
template <typename T>
    class Array
{
public:
    typedef T value_type;

    void set(size_t pos, const value_type &obj) {
        data_[pos] = obj;
    }

    const value_type &get(size_t pos) const {
        return data_[pos];
    }

private:
    value_type *data_;
};
```

Templates - Concepts

```
class Foo {
public:
	void func();
};

class Bar {
public:
	void other_func();
};

template <typename T>
	void temp_func(T obj)
{
	obj.func();
}
```

Templates - Concepts

- Every template has **implicit** requirements for the given types

Templates - Concepts

- Every template has **implicit** requirements for the given types
  - function members
  - data members
  - operators
  - typedefs
  - ...

Templates - Concepts

- Every template has **implicit** requirements for the given types
  - function members
  - data members
  - operators
  - typedefs
  - ...
- Failing these requirements produces ugly compiler errors

Templates - Concepts

- Every template has **implicit** requirements for the given types
  - function members
  - data members
  - operators
  - typedefs
  - ...
- Failing these requirements produces ugly compiler errors
- Concepts document and enforce these requirements

Templates - Concepts

- Every template has **implicit** requirements for the given types
  - function members
  - data members
  - operators
  - typedefs
  - ...
- Failing these requirements produces ugly compiler errors
- Concepts document and enforce these requirements
- Not (yet) part of C++
- anyway good tool to document and communicate the requirements

Outline

TMP Example: Factorial

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

TMP Example: Factorial

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

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

TMP Example: Factorial

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

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

const int some_constant = Factorial<7>::value;
```

TMP Overview

- Programming with Templates

TMP Overview

- Programming with Templates
- Not designed for general purpose programming
- Discovered to be turing complete after standardization

TMP Overview

- Programming with Templates
- Not designed for general purpose programming
- Discovered to be turing complete after standardization
- no variables
- pure functional language

TMP Overview

- Programming with Templates
- Not designed for general purpose programming
- Discovered to be turing complete after standardization
- no variables
- pure functional language
- many language constructs similar to Haskell

$\Rightarrow$ used as pseudocode language

TMP Overview

- Programming with Templates
- Not designed for general purpose programming
- Discovered to be turing complete after standardization
- no variables
- pure functional language
- many language constructs similar to Haskell

$\Rightarrow$ used as pseudocode language

- Boost.Mpl

Outline

Units: Basic Idea

- when calculating with quantities one usually has units

Units: Basic Idea

- when calculating with quantities one usually has units
- only certain operations are allowed (3m + 7s is not defined)

Units: Basic Idea

- when calculating with quantities one usually has units
- only certain operations are allowed ($3\text{m} + 7\text{s}$ is not defined)
- when units of the result match it is likely the calculation is correct

Units: Basic Idea

- when calculating with quantities one usually has units
- only certain operations are allowed (3m $+$ 7s is not defined)
- when units of the result match it is likely the calculation is correct

$\Rightarrow$ let the compiler check the units

Units: Basic Idea

- when calculating with quantities one usually has units
- only certain operations are allowed ($3\text{m} + 7\text{s}$ is not defined)
- when units of the result match it is likely the calculation is correct

$\Rightarrow$ let the compiler check the units

- Idea based on Boost.Unit

Units: International System of Units

- most widely used system of units

Units: International System of Units

- most widely used system of units
- 7 base units:
  - length: metre (m)
  - mass: kilogramm (kg)
  - time: second (s)
  - electric current: ampere (A)
  - thermodynamic temperature: kelvin (K)
  - luminous intensity: candela (cd)
  - amount of substance: mole (mol)

Units: International System of Units

- most widely used system of units
- 7 base units:
  - length: metre (m)
  - mass: kilogramm (kg)
  - time: second (s)
  - electric current: ampere (A)
  - thermodynamic temperature: kelvin (K)
  - luminous intensity: candela (cd)
  - amount of substance: mole (mol)
- All other units are derived from base units
  - force: newton ($N = \frac{kg \cdot m}{s^2}$)
  - electric resistance: ohm ($\Omega = \frac{kg \cdot m^2}{s^2 \cdot A^2}$)
  - ...

Units: International System of Units

- most widely used system of units
- 7 base units:
  - length: metre (m)
  - mass: kilogramm (kg)
  - time: second (s)
  - electric current: ampere (A)
  - thermodynamic temperature: kelvin (K)
  - luminous intensity: candela (cd)
  - amount of substance: mole (mol)
- All other units are derived from base units
  - force: newton ($N = \frac{kg \cdot m}{s^2}$)
  - electric resistance: ohm ($\Omega = \frac{kg \cdot m^2}{s^2 \cdot A^2}$)
  - ...

Units: Interface

- Express units by types

Units: Interface

- Express units by types
- use **int** template parameters to express exponentiation of base units

Units: Interface

- Express units by types
- use **int** template parameters to express exponentiation of base units

```
Length l = 25.35 * metre;
Mass m   = 19 * kilogramm;
Time t1  = 12 * second;
Time t2  = 19 * second;

Force f = l * m / (t1 * t2);
```

Units: Interface

- Express units by types
- use **int** template parameters to express exponentiation of base units

```
Length l = 25.35 * metre;
Mass m   = 19 * kilogramm;
Time t1  = 12 * second;
Time t2  = 19 * second;

Force f = l * m / (t1 * t2);
```

```
Length l = 17.49 * second;
```

Units: Interface

- Express units by types
- use **int** template parameters to express exponentiation of base units

```
Length l = 25.35 * metre;
Mass m   = 19 * kilogramm;
Time t1  = 12 * second;
Time t2  = 19 * second;

Force f = l * m / (t1 * t2);
```

```
Length l = 17.49 * second;
```

ERROR!

Units: Code - class `Unit`

```
template <int M, int KG, int S>
    class Unit
{
public:
    Unit(double value)
    : value_(value) {}

    double value() const {
        return value_;
    }

private:
    double value_;
};
```

Units: Code - Operators (1/3)

```
template <int M, int KG, int S>
      Unit<M, KG, S>
      operator*(double lhs, Unit<M, KG, S> rhs)
{
      return lhs * rhs.value();
}

template <int M, int KG, int S>
      Unit<M, KG, S>
      operator*(Unit<M, KG, S> lhs, double rhs)
{
      return lhs.value() * rhs;
}
```

Units: Code - Operators (2/3)

```
template <
        int M1, int KG1, int S1,
        int M2, int KG2, int S2
>
        Unit<
                M1 + M2,
                KG1 + KG2,
                S1 + S2
        >
        operator*(
                Unit<M1, KG1, S1> lhs,
                Unit<M2, KG2, S2> rhs
        )
{
        return lhs.value() * rhs.value();
}
```

Units: Code - Operators (3/3)

```
template <
      int M1, int KG1, int S1,
      int M2, int KG2, int S2
>
      Unit<
            M1 - M2,
            KG1 - KG2,
            S1 - S2
      >
      operator/(
            Unit<M1, KG1, S1> lhs,
            Unit<M2, KG2, S2> rhs
      )
{
      return lhs.value() / rhs.value();
}
```

Units: Code - typedefs and Objects

```
typedef Unit<1, 0, 0> Length;
typedef Unit<0, 1, 0> Mass;
typedef Unit<0, 0, 1> Time;

typedef Unit<1, 1, -2> Force;

Length metre(1.0);
Mass kilogramm(1.0);
Time second(1.0);

Force newton(1.0);
```

Units - Conclusion

- Only proof of concept code
- Lots of work remains to be done

Units - Conclusion

- Only proof of concept code
- Lots of work remains to be done
- If you want to use this code $\Rightarrow$ Boost.Unit

DSEL

- Domain Specific Embedded Language

DSEL

- Domain Specific Embedded Language
- DSL: A language specially designed for a specific purpose

DSEL

- Domain Specific Embedded Language
- DSL: A language specially designed for a specific purpose
- DSEL: A DSL embedded into C++

DSEL

- Domain Specific Embedded Language
- DSL: A language specially designed for a specific purpose
- DSEL: A DSL embedded into C++
- Implemented using operator overloading and TMP

DSEL

- Domain Specific Embedded Language
- DSL: A language specially designed for a specific purpose
- DSEL: A DSL embedded into C++
- Implemented using operator overloading and TMP
- Boost.Proto

Boost.Spirit

- Parser generator for context free grammars

Boost.Spirit

- Parser generator for context free grammars
- DSL: EBNF
- DSEL: as close to EBNF as possible

Boost.Spirit

- Parser generator for context free grammars
- DSL: EBNF
- DSEL: as close to EBNF as possible
- Because of missing operators/operator precedence: Little differences

Boost.Spirit: Sample

- EBNF

```
start      = expression | addition ;
expression = "(" , addition , ")" | number ;
addition   = expression , "+" , expression ;
(* number is omitted here *)
```

Boost.Spirit: Sample

- EBNF

```
start      = expression | addition ;
expression = "(" , addition , ")" | number ;
addition   = expression , "+" , expression ;
(* number is omitted here *)
```

- Boost.Spirit

```
start      = expression | addition;
expression = '(' >> addition >> ')' | int_;
addition   = expression >> '+' >> expression;
// int_ is provided by Spirit
```

Boost.Spirit

- Based on PEG

Boost.Spirit

- Based on PEG
- Can generate parsers, writers and lexers

Boost.Spirit

- Based on PEG
- Can generate parsers, writers and lexers
- Many possibilities to access the parsed data

Boost.Spirit

- Based on PEG
- Can generate parsers, writers and lexers
- Many possibilities to access the parsed data
  - semantic actions

Boost.Spirit

- Based on PEG
- Can generate parsers, writers and lexers
- Many possibilities to access the parsed data
  - semantic actions
  - attributes

Boost.Spirit

- Based on PEG
- Can generate parsers, writers and lexers
- Many possibilities to access the parsed data
  - semantic actions
  - attributes
  - parse tree (utree)

Boost.Parameter

- C++ has no named function arguments

Boost.Parameter

- C++ has no named function arguments
- calling functions with many parameters can be annoying and error prone

Boost.Parameter

- C++ has no named function arguments
- calling functions with many parameters can be annoying and error prone

⇒ Building named function arguments as a library:

```
print_text(
      "the text",
      font_size=12,
      bold=true,
      italic=false,
      color=green
);
```

Expression Templates

- a + b + c yields code like add(add(a, b), c)

Expression Templates

- a + b + c yields code like add(add(a, b), c)

⇒ add(a, b) returns a temporary object which is only used to add c

Expression Templates

- a + b + c yields code like add(add(a, b), c)

⇒ add(a, b) returns a temporary object which is only used to add c

- if you add large matrices this can become a performance problem

Expression Templates

- a + b + c yields code like add(add(a, b), c)

⇒ add(a, b) returns a temporary object which is only used to add c

- if you add large matrices this can become a performance problem
- use TMP to evaluate the whole expression when the result is required

Expression Templates

- a + b + c yields code like add(add(a, b), c)

⇒ add(a, b) returns a temporary object which is only used to add c

- if you add large matrices this can become a performance problem
- use TMP to evaluate the whole expression when the result is required
- enables generation of highly optimised code while enabling the programmer to use a natural syntax
- performs similar to Fortran

Expression Templates

- a + b + c yields code like add(add(a, b), c)

⇒ add(a, b) returns a temporary object which is only used to add c

- if you add large matrices this can become a performance problem
- use TMP to evaluate the whole expression when the result is required
- enables generation of highly optimised code while enabling the programmer to use a natural syntax
- performs similar to Fortran
- sample: blitz++

Summary

- areas of application

Summary

- areas of application
  - type safety

Summary

- areas of application
  - type safety
  - missing language features

Summary

- areas of application
  - type safety
  - missing language features
  - DSEL

Summary

- areas of application
  - type safety
  - missing language features
  - DSEL
  - performance

Summary

- areas of application
  - type safety
  - missing language features
  - DSEL
  - performance
- limitations

Summary

- areas of application
  - type safety
  - missing language features
  - DSEL
  - performance
- limitations
  - confusing compiler error messages

Summary

- areas of application
  - type safety
  - missing language features
  - DSEL
  - performance
- limitations
  - confusing compiler error messages
  - long compile times

Summary

- areas of application
  - type safety
  - missing language features
  - DSEL
  - performance
- limitations
  - confusing compiler error messages
  - long compile times
  - the code can become a bit obscure

Thank you for your attention!
questions?