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
- @rioderelfte on twitter
- http://www.r9e.de
Outline

Metaprogramming

C++

 Templates

Introduction To Template Metaprogramming

TMP considered sexy
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

```c
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

Metaprogramming

C++

Templates

Introduction To Template Metaprogramming

TMP considered sexy
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

- Boost: A set of high quality C++ libraries. Many of them utilise or provide TMP
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
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

Metaprogramming

C++

Templates

Introduction To Template Metaprogramming

TMP considered sexy
Templates - Template Function max

```cpp
10 int main() {
11    std::cout << max(23, 42) << std::endl
12        << max('c', '+') << std::endl;
13 }
```
Templates - Template Function max

1 template <typename T>
2 T max(T a, T b)
3 {
4     return (a > b)
5     ? a : b;
6 }
7
8
9

10 int main() {
11     std::cout << max(23, 42) << std::endl
12         << max('c', '+') << std::endl;
13 }
Templates - Template Function \texttt{max}

\begin{verbatim}
1 template <typename T> int max(int a, int b) {
2     T max(T a, T b) {
3         return (a > b)? a : b;
4     }
5     return (a > b)? a : b;
6 }
7 char max(char a, char b) {
8     return (a > b)? a : b;
9 }
10 int main() {
11     std::cout << max(23, 42) << std::endl
12         << max('c', '+') << std::endl;
13 }
\end{verbatim}
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
```cpp
// 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

```cpp
1 class Foo {
2 public:
3     void func();
4 }
5
6 class Bar {
7 public:
8     void other_func();
9 }
10
11 template <typename T>
12     void temp_func(T obj)
13     {
14         obj.func();
15     }
```
Templates - Concepts

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

Metaprogramming

C++

Templates

Introduction To Template Metaprogramming

TMP considered sexy
TMP Example: Factorial

```cpp
1 template <int N>
2 struct Factorial
3 {
4     static const int value =
5         N * Factorial<N-1>::value;
6 }
```
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

```cpp
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

Boost.Mpl
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

⇒ used as pseudocode language
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
⇒ used as pseudocode language
Boost.Mpl
Outline

Metaprogramming

C++

Templates

Introduction To Template Metaprogramming

TMP considered sexy
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 (3m + 7s 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
  ⇒ let the compiler check the units
Units: Basic Idea

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

⇒ 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 (Ω = $\frac{kg \cdot m^2}{s^2 \cdot A^2}$)
  - ...
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

1. Length $l = 25.35 \times \text{metre}$;
2. Mass $m = 19 \times \text{kilogramm}$;
3. Time $t1 = 12 \times \text{second}$;
4. Time $t2 = 19 \times \text{second}$;
5. Force $f = l \times \frac{m}{(t1 \times t2)}$;
Units: Interface

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

```
1 Length l = 25.35 * metre;
2 Mass m = 19 * kilogramm;
3 Time t1 = 12 * second;
4 Time t2 = 19 * second;
5
6 Force f = l * m / (t1 * t2);
```

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

Units: Interface

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

1. Length \( l = 25.35 \times \text{metre} \);
2. Mass \( m = 19 \times \text{kilogramm} \);
3. Time \( t_1 = 12 \times \text{second} \);
4. Time \( t_2 = 19 \times \text{second} \);
5. Force \( f = l \times m / (t_1 \times t_2) \);

1. Length \( l = 17.49 \times \text{second} \)  \( \text{ERROR!} \)
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)

1 template <int M, int KG, int S>
2     Unit<M, KG, S>
3     operator*(double lhs, Unit<M, KG, S> rhs)
4     {
5         return lhs * rhs.value();
6     }
7
8 template <int M, int KG, int S>
9     Unit<M, KG, S>
10     operator*(Unit<M, KG, S> lhs, double rhs)
11     {
12         return lhs.value() * rhs;
13     }
```
```cpp
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();
}
```
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();
}
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

Boost.Proto
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

1. `start := expression | addition;`
2. `expression := "(", addition, ")" | number;`
3. `addition := expression, "+", expression;`
4. `(* number is omitted here *)`
Boost.Spirit: Sample

- EBNF

```
1 start  ::= expression | addition ;
2 expression ::= " (" , addition , ")" | number ;
3 addition  ::= expression , "+" , expression ;
4 (* number is omitted here *)
```

- Boost.Spirit

```
1 start  ::= expression | addition ;
2 expression ::= ' ( ' >> addition >> ' ) ' | int_;
3 addition  ::= expression >> '+' >> expression ;
4 // 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

```cpp
call(text="the text", font_size=12, bold=true, italic=false, color=green)
```
C++ has no named function arguments

- calling functions with many parameters can be annoying and error prone

⇒ Building named function arguments as a library:

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

- $a + b + c$ yields code like $\text{add}(\text{add}(a, b), c)$

- $\text{add}(a, b)$ returns a temporary object which is only used to add $c$

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

- Use $\text{TMP}$ 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++
Expression Templates

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

- sample: blitz++
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 $\text{add}(\text{add}(a, b), c)$

⇒ $\text{add}(a, b)$ 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?