Template Metaprogramming in C++

Krzysztof Czarnecki, DaimlerChrysler AG
czarnecki@acm.org

&

Ulrich W. Eisenecker, FH Kaiserslauten
ulrich.eisenecker@t-online.de

Overview

- *What is template metaprogramming?*
- Metainformation
- Computing values
- Computing types
- Code generation
- Outlook and valuation
- References

Metaprogramming

- Writing programs that represent and manipulate other programs or themselves
- Metaprograms are programs about other programs or themselves, e.g., compilers, interpreters, macros …
- Metaprograms can be executed at compile time (static) or runtime (dynamic)

Template Metaprogramming

- Template metaprograms are executed by the compiler at compile time (static metaprogramming)
- Representing and manipulating programs (classic metaprogramming)
- Static computation of values

Overview

- What is template metaprogramming?
- *Metainformation*
- Computing values
- Computing types
- Code generation
- Outlook and valuation
- References

Metainformation

Information on types

- In a type itself (as its member)
- Using a traits template
- Using a traits class

Metainformation

In a type itself

```
struct Int
{    enum { is_integral = 1,is_exact = 1 };
     // …
};
struct Double
{    enum { is_integral = 0, is_exact = 0 };
     // …
};

template <class IntegralsOnly> void foo(const IntegralsOnly& I)
{    assert (IntegralsOnly::is_integral == true);
     // …
}
```

Can cause a run time Assertion failure!
Good? Bad?
Alternatives?

Metainformation

In a type itself

+ Information and metainformation located in one place
– Metainformation cannot be extended without invasively modifying the implementation of the type, i.e., no metainformation for basic types
– Metainformation for various purposes is mixed

Metainformation

Traits templates

```
template<class T>
class numeric_limits
{    public:
          static const bool is_specialized = false;
          static T min() throw();
          static T max() throw();
          static const int  digits = 0;
          static const int  digits10 = 0;
          static const bool is_signed = false;
          static const bool is_integer = false;
          static const bool is_exact = false;
          static const int  radix = 0;
          static T epsilon() throw();
          static T round_error() throw();
          // ...
```

Excerpt from numeric_limits

Metainformation

Traits templates

```
class numeric_limits<float>
{    public:
        static const bool is_specialized = true;
        inline static float min() throw() { return 1.17549435E-38F; }
        inline static float max() throw() { return 3.40282347E+38F; }
        static const int digits   = 24;
        static const int digits10 =  6;
        static const bool is_signed  = true;
        static const bool is_integer = false;
        static const bool is_exact   = false;
        static const int radix = 2;
        inline static float epsilon() throw()     { return 1.19209290E-07F; }
        inline static float round_error() throw() { return 0.5F; }
        // ...
```

Excerpt from the specialization for numeric_limits<float>

Metainformation

Traits templates

```
cout << numeric_limits<float>::is_integer << endl;
cout << numeric_limits<int>::is_integer << endl;

cout << numeric_limits<complex<float> >::is_specialized << endl;
// “false” would imply that numeric_limits was not specialized
// for complex<float>
```

Usage

Metainformation

Traits templates

+ Extending metainformation without modifying the type
+ Modularly dividing metainformation (numeric_limits, special_metainfo ...)
– Only one specialization of a traits template per type
– Information and metainformation located in different places

Metainformation

Traits classes (configuration repository)

```
struct SampleConfig
{   typedef double Element;
    typedef unsigned short Size;
    enum { size = 5 };
    typedef Vector<SampleConfig> SCVector;
};
```

Metainformation

Using Traits classes

```
template <class config> class Vector
{    public:
          typedef config Config; // export config for "clients"
          typedef typename Config::Element Element; // Define "local short hands"
          typedef typename Config::Size Size;
          enum {thesize = Config:size};
          Vector() :size_(thesize)
          { for (Size i=0;i<size();++i) el[i] = 0.0; }
          const Size& size() const
          { return size_; }
          void element(const Size& i,const Element& e)
          { el[i] = e; }
          const Element& element(const Size& i) const
          { return el[i]; }
     private:
          Element el[thesize];
          const int size_=thesize;
};
```

Metainformation

```
typedef SampleConfig::SCVector Vector;
Vector v;
v.element(2,42.1);
cout << v.size();
typedef Vector::Size Size;
for (Size i=0;i<v.size();++i)
cout << '\t' << v.element(i);
cout << endl;
```

Metainformation

```
struct SampleConfig
{   struct ForVector
    {   typedef int Element;
        enum { size = 5 };
    };
    struct ForMatrix
    {   typedef double Element;
        enum { rows = 5, cols = 5 };
    };
    typedef Vector<SampleConfig> SCVector;
    typedef Matrix<SampleConfig> SCMatrix;
};
```

Metainformation

```
template <class config> class Vector
{    public:
            typedef config Config; // export config
            typedef typename Config::ForVector MyConfig;
            typedef typename MyConfig::Element Element;
     // ...
};

template <class config> class Matrix
{    public:
            typedef config Config; // export config
            typedef typename config::ForMatrix MyConfig;
            typedef typename MyConfig::Element Element;
     // ...
};
```

Metainformation

Structured configuration repository

```
typedef SampleConfig::SCVector Vector;
typedef SampleConfig::SCMatrix Matrix;
Vector v;
Matrix m;
// ...
```

Metainformation

Computing configuration repositories

```
struct Precise
{    typedef long double Element; };

struct Compact
{    typedef float Element; };

template <int Size = 10, class Precision = Compact>
struct ComputeConfig
{    struct Config
     {     typedef typename Precision::Element Element;
           enum { size = Size };
           typedef Vector<Config> CCVector;
     };
};
```

Metainformation

Computing configuration repositories

```
template <class config> class Vector
{   public:
        typedef config Config; // export config
        typedef typename Config::Element Element;
        // ...
};

typedef ComputeConfig<5,Precise>::Config SampleConfig;
typedef SampleConfig::CCVector Vector;
Vector v;
// ...
```

Metainformation

Traits classes

+ Parameterizing components with different traits classes in a flexible way
+ Traits classes containing configuration information for a set of components (configuration repository)
+ Structured configuration repositories
+ Computing configuration repositories
– Information and metainformation located in different places
– Configuration repositories can get large

Overview

- What is template metaprogramming?
- Metainformation
- *Computing values*
- Computing types
- Code generation
- Outlook and valuation
- References

Computing Values

- Computing constants
- Simple template metafunctions (TMF)
- TMF calling other TMF
- TMF with several statements
- TMF as TMF parameters

Computing Values

Computing constants

```
const double PI = 22.0/7.0;
const double PI_SQUARED = PI*PI;
```

Advantages

- More descriptive (higher intentionality)
- Implementation (the computation formula) easy to modify

Computing Values

Computing constants

```
// not descriptive, hard to maintain - value fixed at compile time
const int C1 = 40320;

// more descriptive, hard to maintain - computed at compile time
const int C2 = 1*2*3*4*5*6*7*8;

int factorial(int n)
{   return (n == 0) ? 1 : n * factorial(n-1);
}
// descriptive, easy to maintain - computed at runtime
const int C3 = factorial(8);
```

Computing Values

Simple template metafunctions (TMF)

```
template <int n>
struct Factorial
{   enum { RET = n * Factorial<n-1>::RET };
};

template <>
struct Factorial<0>
{   enum { RET = 1 };
};

// descriptive, easy to maintain, computed at compile time
cout << Factorial<8>::RET << endl; //   cout << 40320 << endl;
```

Computing Values

```
template <3>
struct Factorial
{   enum { RET = 3 * Factorial<2>::RET }; // RET = 3*2*1*1
};
    template <2>
    struct Factorial
    {     enum { RET = 2 * Factorial<1>::RET }; // RET = 2*1*1
    };
          template <1>
          struct Factorial
          {         enum { RET = 1 * Factorial<0>::RET }; // RET = 1*1
          };
                    template <>
                    struct Factorial<0>
                    {         enum { RET = 1 }; // RET = 1
                    };
```

Computing Values

TMF calling other TMF

- Take k from n elements
  - without putting elements back
  - without taking order into account

(Combinations without repetitions)

$$C(k,n) = \frac{n!}{k! \cdot (n - k)!}$$

Computing Values

TMF calling other TMF

```
template<int k, int n>
struct Combinations
{   enum
    { RET =    Factorial<n>::RET /
               (Factorial<k>::RET * Factorial<n-k>::RET) };
};

// ...
cout << Combinations<2,4>::RET << endl;
```

Computing Values

TMF with several statements

```
template<int k, int n>
class Combinations
{       enum
        {   num = Factorial<n>::RET,            //statement 1
            denom = Factorial<k>::RET *         //statement 2
            Factorial<n-k>::RET
        };
    public:
        enum { RET = num / denom };             // statement 3
};
```

Computing Values

TMF as TMF parameters

```
// accumulate(n,f) := f(0) + f(1) + … + f(n)

// Passing a template as a template parameter
template<int n, template<int> class F>
struct Accumulate
{ enum { RET = Accumulate<n-1,F>::RET + F<n>::RET };
};

template<template<int> class F>
struct Accumulate<0,F>
{ enum { RET = F<0>::RET };
};
```

Computing Values

TMF as TMF parameters

```
template<int n>
struct Square
{ enum { RET = n*n };
};

// ...
cout << Accumulate<3,Square>::RET << endl;
```

Computing Values

TMF as TMF parameters

```
struct Square // Passing as a parameter using a member template
{ template<int n> struct Apply
  {  enum { RET = n*n };
  };
};

template<int n, class F>
struct Accumulate
{  enum { RET = Accumulate<n-1,F>::RET + F::template Apply<n>::RET };
};

template<class F>
struct Accumulate<0,F>
{  enum { RET = F::template Apply<0>::RET };
};
```

Computing Values

+ Computation at compile time, thus
  - better performance
  - smaller object code
+ Descriptive and easy to maintain representations of constants
+ Full spectrum as for dynamic function calls
  - Functions calling other functions
  - Functions as function parameters
– Recursion as the only iteration mechanism
– Unusual syntax
– Longer compilation times

Overview

- What is template metaprogramming?
- Metainformation
- Computing values
- *Computing types*
- Code generation
- Outlook and valuation
- References

Computing types

- Simple selection
- Multiple selection
- Decision tables

Computing types

Simple selection - application example

- Vector<int> + Vector<float> □ Vector<float>
- Solution: TMF selecting the conceptually larger type
- The necessary metainformation available (numeric_limits<>::max_exponent10, numeric_limits<>::digits)
- Required: A simple selection of one from two types based on a Boolean expression

Computing types

Simple selection - application example

```
template<bool condition, class Then, class Else>
struct IF
{ typedef Then RET;
};

// partial specialization
template<class Then, class Else>
struct IF<false, Then, Else>
{ typedef Else RET;
};

//...
IF<(1+2>4), int, float>::RET i; // I is of type float.
```

Computing types

Simple selection - application example

```
template<class A, class B>
struct Promote
{   enum
    { cond = numeric_limits<A>::max_exponent10 <
      numeric_limits<B>::max_exponent10 ||
      ( numeric_limits<A>::max_exponent10 ==
          numeric_limits<B>::max_exponent10 &&
          numeric_limits<A>::digits <
          numeric_limits<B>::digits )
    };

    typedef typename IF<cond,B,A>::RET RET;
};
```

Computing types

Simple selection - application example

```
template <class T1,class T2> Matrix<Promote<T1,T2>::RET>
operator+(Matrix<T1>& m1,Matrix<T2>& m2)
{ Matrix<Promote<T1,T2>::RET> result;
   // ... Addition of Elements
   return result;
}

//…
typedef int Type1;
typedef double Type2;
Matrix<Type1> m;
Matrix<Type2> n;
Matrix<Promote<Type1,Type2>::RET> r = m + n;
```

Computing types

Simple selection - an alternative

```
// Implementation using member templates

  // class with a member template for selecting ThenType
  struct SelectThen
  { template<class ThenType, class ElseType>
    struct Result
    { typedef ThenType RET; };
  };

  // class with a member template for selecting ElseType
  struct SelectElse
  { template<class ThenType, class ElseType>
    struct Result
    { typedef ElseType RET; };
  };
```

Computing types

```
template<bool condition> // Choosing a selecting class
  struct ChooseSelector
  { typedef SelectThen RET; };

  template<>
  struct ChooseSelector<false>
  { typedef SelectElse RET; };

template<bool condition, class ThenType, class ElseType>
struct IF
{   typedef typename ChooseSelector<condition>::RET Selector;
    typedef typename Selector::template Result<ThenType,ElseType>::RET RET;
};
```

Computing types

Simple selection

- Used like a “normal” if
- IF and its evaluation are easy to save
  - typedef IF< 1!=2, A, B> Result;
  - typedef IF< 1!=2, A, B>::RET Result;
- Type-dependent selection of code or member access
  - IF< 1!=2, A, B>::RET::some_static_function();
  - int result = IF< 1!=2, A, B>::RET::some_int;

Computing types

Multiple selection - application example

- Customization of a automatic teller machine (ATM)
- Messages in a local language depending on the country of installation
- A given ATM contains only the code for the needed language
- Configuration at compile time

Computing types

Multiple selection - application example

```
struct ATMGerman
{   static const char* creditOverdraft()
    {    return "Kreditrahmen ueberschritten"; }
    static const char* limitOverdraft()
    {    return "Hoechstbetrag ueberschritten"; }
    static const char* insufficientCash()
    {    return "Unzureichender Geldvorrat"; }
    static const char* otherMalfunction()
    {    return "Funktionsstoerung"; }
};

// ATMEnglisch, ATMPolish
```

Computing types

Multiple selection - application example

```
template <class Language>
class ATM: protected Language
{   private:
        typedef ATM< Language > self;
    public:
        // ...
        void withdraw(const double& amount)
        {   if (isCreditOverdraft(amount))
            {   cout << self::creditOverdraft() << endl;
                return;
            }
            if (amount > limit())
            {   cout << self::limitOverdraft() << endl;
                return;
            }
            // ...
```

Computing types

Multiple selection - application example

```
enum Country { Deutschland, England, Oesterreich, Polska, USA };
// ...
LOCALIZE_ATM<Deutschland>::RET atm;
test(atm);
```

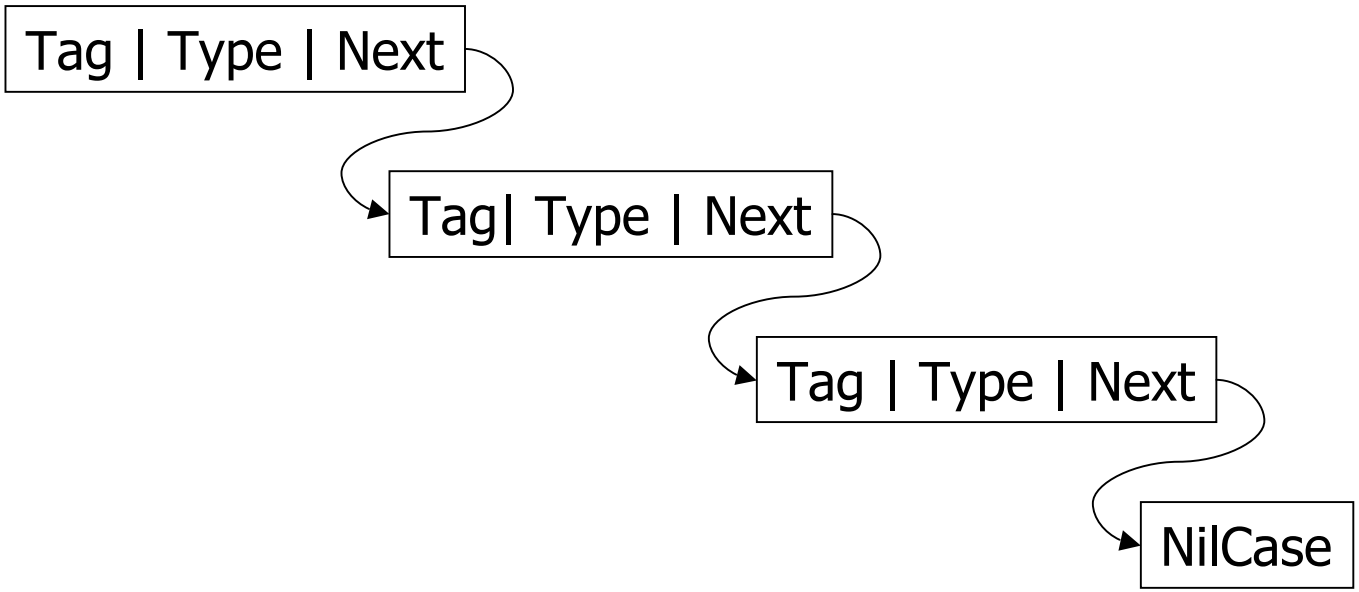
Computing types

Multiple selection - application example

```
template <Country country>
struct LOCALIZE_ATM
{    typedef typename
          SWITCH<country,
               CASE<Deutschland,ATMGerman,
               CASE<England,ATMEnglisch,
               CASE<Oesterreich,ATMGerman,
               CASE<Polska,ATMPolnish,
               CASE<USA,ATMEnglisch
          > > > > > >::RET Language;
     typedef ATM<Language> RET;
};
```

Computing types

Multiple selection - implementation



Computing types

Multiple selection - implementation

```
const int DEFAULT = ~(~0u >> 1); // platform independent most negative int value

struct NilCase
{};

template <int tag_, class Type_, class Next_ = NilCase>
struct CASE
{    enum { tag = tag_ };
     typedef Type_ Type;
     typedef Next_ Next;
};
```

Computing types

Multiple selection - implementation

```
template<int tag, class Case> // Version with partial specialization
struct SWITCH
{   typedef typename Case::Next NextCase;
    enum
    {   caseTag = Case::tag,
        found   = (caseTag == tag || caseTag == DEFAULT)
    };
    typedef typename
        IF<found,
            typename Case::Type,
            typename SWITCH<tag, NextCase>::RET
        >::RET RET;
};

template<int tag>
struct SWITCH<tag, NilCase>
{   typedef NilCase RET;
};
```

Computing types

Multiple selection

- Usage like a “normal” switch; different implementations possible
  - DEFAULT<…> instead of CASE<DEFAULT, …>
  - DEFAULT<…> can stand at an arbitrary place
- Implementation with member templates possible
- Special checks
  - Detecting multiple DEFAULT branches
  - Detecting multiple tags

Computing types

Dependency analysis - application example

- Customizing output language based on country and state
- One ATM contains only the code for the needed language
- Configuration at compile time

Computing types

CountryState**Language**Deutschland***ATM**German**England***ATM**

Computing types

Dependency table - usage

```
typedef typename EVAL_TABLE
//*************************************************************************
<        CELL<     ountry    , CELL<  state                                  > >
//-------------------------------------------------------------------------------------
, ROW< CELL<Deutschland , CELL<  anyValue      , ATMGerman                  > >
, ROW< CELL<   England   , CELL<  anyValue      , ATMEnglish                 > >
, ROW< CELL< Oesterreich , CELL<  anyValue      , ATMGerman                  > >
, ROW< CELL<     Polska   , CELL<  anyValue      , ATMPolnish                 > >
, ROW< CELL<      USA     , CELL<  anyValue      , ATMEnglish                 > >
, ROW< CELL<   Canada    , CELL<   Quebec       , ATMFrench                  > >
, ROW< CELL<   Canada    , CELL<  anyValue      , ATMEnglisch                > >
, ROW< CELL<   Schweiz   , CELL<     Jura        , ATMFrench                  > >
, ROW< CELL<   Schweiz   , CELL<    Wallis       , ATMFrench                  > >
, ROW< CELL<   Schweiz   , CELL<      Genf       , ATMFrench                  > >
, ROW< CELL<   Schweiz   , CELL<   Freiburg      , ATMFrench                  > >
, ROW< CELL<   Schweiz   , CELL<    Waadt        , ATMFrench                  > >
, ROW< CELL<   Schweiz   , CELL<    Tessin       , ATMItalienisch             > >
, ROW< CELL<   Schweiz   , CELL<   Zuerich       , ATMGerman                  > >
, ROW< CELL< anyValue    , CELL<  anyValue      , ATMEnglisch                > >
//*************************************************************************
> > > > > > > > > > > > > > > > >::RET Language;
```

Computing types

Dependency table - usage

```
enum Country
{    Deutschland,  // etc.
};

enum State
{    // Deutschland
     BadenWuerttemberg,  // etc.
};

template <Country country, State state>
struct LOCALIZE_ATM
{    typedef typename EVAL_TABLE
     // the rest as on the previous slide
     typedef ATM<Language> RET;
};

LOCALIZE_ATM<Schweiz,Wallis>::RET atm;
test(atm);
```

Computing types

Dependency tables

- Implementation somewhat more complex (see Knaupp, Eisenecker, Czarnecki: Mit Tabellen zur Entscheidung, in OBJEKTspektrum 5/99 or the "Generative Programming" book)
- Extended dependency tables
  - Evaluation in the top-down direction
  - joker (anyValue)
- Returning multiple values
  - Chaining results in lists
  - Configuration repositories

Overview

- What is template metaprogramming?
- Metainformation
- Computing values
- Computing types
- *Code generation*
- Outlook and valuation
- References

Code Generation

- Type-dependent selection of code
  - IF< StaticCondition, CodeContainerA, CodeContainerB>::RET::some_static_function();
- Together with inlining, the above statement will be reduced to the desired code.
- Some_static_function is usually called "exec" ☺

Code Generation

- Code-generating TMF
- DO
- WHILE
- FOR

Code Generation

Code generation - application example

- A number of localizations for our ATM are available
- The ATM should be tested for each localization

Code Generation

Explicit encoding of all tests...

```
// redundant, hard to maintain, and error-prone...
{   LOCALIZE_ATM<Deutschland>::RET atm;
    test(atm);
}
{   LOCALIZE_ATM <England>::RET atm;
    test(atm);
}
// ...
```

Code Generation

Alternative: the use of a code generating TMF...

```
enum Country { Deutschland, England, Oesterreich, Polska, USA, STOP };
// LOCALIZE_ATM as before

template <Country country = Deutschland>
struct testAll
{    static void execute()
     {     LOCALIZE_ATM<country>::RET atm;
           test(atm);
           typedef testAll<Country(country + 1)> nextTest;
           nextTest::execute();
     }
};

template <> struct testAll<STOP> // template specialization for STOP terminates the recursion
{    static void execute() {};
};

// ...
testAll<>::execute();
```

Code Generation

DO & WHILE - usage

```
enum Country { Deutschland, England, Oesterreich, Polska, USA }; // STOP not needed!
// LOCALIZE_ATM as before

template <Country country = Deutschland>
struct Statement
{       enum { country_ = country };
        static void exec()
        {       LOCALIZE_ATM<country>::RET atm;
                test(atm);
        }
        typedef Statement<Country(country + 1)> Next;
};

struct EndCondition
{       template <class Statement_>
        struct Code
        {       enum { RET = Country(Statement_::country_) <= USA };
        };
};


// ...
DO<Statement<>,EndCondition>::exec();
WHILE<EndCondition,Statement<> >::exec();
```

Code Generation

DO - Implementation

```
struct Stop
{    static void exec() {};
};

template <class Statement, class Condition>
struct DO
{    typedef typename Statement::Next NewStatement;
     static void exec()
     {     Statement::exec();
           IF   <Condition::template Code<NewStatement>::RET,
                     DO<NewStatement,Condition>,
                     Stop
                >::RET::exec();
     }
};
```

Code Generation

WHILE - Implementation

```
struct Stop
{    static void exec() {};
};

template <class Condition, class Statement>
struct WHILE
{    static void exec()
     {     IF   <Condition::template Code<Statement>::RET,
                    Statement,
                    Stop
               >::RET::exec();
          typedef typename Statement::Next NewStatement;
          IF   <Condition::template Code<Statement>::RET,
                    WHILE<Condition, NewStatement >,
                    Stop
               >::RET::exec();
     }
};
```

Code Generation

FOR - Usage

```
struct Statement
{   template <int country>
    struct Code
    {   static void exec()
        {   LOCALIZE_ATM<(Country)country>::RET atm;
            test(atm);
        }
    };
};

// ...
FOR<Deutschland,LessEqual,USA,1,Statement>::exec();
```

Code Generation

FOR - Implementation

```
// Relational TMF
struct Less
{    template<int x, int y>
     struct Code
     { enum { RET = x<y };
     };
};

struct LessEqual
{    template<int x, int y>
     struct Code
     { enum { RET = x<=y };
     };
};

// ...
```

Code Generation

FOR - Implementation

```
template <int from, class Compare, int to, int by, class Statement>
struct FOR
{   static void exec()
    {   typedef typename Statement::Code<from> Code_;
        IF  <Compare::template Code<from,to>::RET,
                Code_,
                Stop
            >::RET::exec();
        IF  <Compare::template Code<from,to>::RET,
                FOR<from+by,Compare,to,by,Statement>,
                Stop
            >::RET::exec();
    }
};
```

Code Generation

FOR - nesting loops

```
int m[3][4] =  {     { 1, 2, 3, 4},
                     { 5, 6, 7, 8},
                     { 9,10,11,12}
                     };

struct OuterLoop
{     template <int i>
      struct Code
      {       struct InnerLoop
              {       template <int j>
                      struct Code
                      {       static void exec()
                              {       cout << i << ',' << j << ": " << m[i][j];
                                      if (j == 3)
                                      cout << endl;
                                      else cout << '\t';
                              }
                      };
              };
              static void exec()
              {       FOR<0,LessEqual,3,+1,InnerLoop>::exec();
              }
      };
};
// ...
FOR<0,LessEqual,2,+1,OuterLoop>::exec();
```

Code Generation

- TMF uses additional compile time, and may increase code-size…but may provide better run-time performance.
- DO, WHILE and FOR can provide greater readability/transparency
- DO, WHILE und FOR can allow direct translation of run-time algorithms.
- DO, WHILE and FOR have recursive implementations, but this is hidden.

Overview

- What is template metaprogramming?
- Metainformation
- Computing values
- Computing types
- Code generation
- *Outlook and valuation*
- References