

Static Metaprogramming in C++

Michael Pisula

Advanced seminar: Modern programming techniques

Outline

- 1 Introduction
- 2 Programming with templates
 - Example
 - Techniques
 - Functional elements in Template C++
- 3 Real-life examples
 - Loki
 - Libraries using Template C++
- 4 Review and Summary

What is Static Metaprogramming

Definition (Static Metaprogramming)

Creation of *Metaprograms* which are evaluated at compile-time

Definition (Template Metaprogramming)

Static Metaprogramming in C++ is also called Template Metaprogramming.

What is Static Metaprogramming

Definition (Static Metaprogramming)

Creation of *Metaprograms* which are evaluated at compile-time

Definition (Template Metaprogramming)

Static Metaprogramming in C++ is also called Template Metaprogramming.

What are Templates

- Templates are classes or functions without specific types
- Originally introduced for generic programming
- Erwin Unruh was to first to use templates for compile-time programming 1994
- Todd Veldhuizen made the first template metaprograms 1995

Example

```
template<typename T>
T max(T x, T y) {
    if(x < y)
        return y;
    else
        return x;
}
```

What are Templates

- Templates are classes or functions without specific types
- Originally introduced for generic programming
- Erwin Unruh was to first to use templates for compile-time programming 1994
- Todd Veldhuizen made the first template metaprograms 1995

Example

```
template<typename T>
T max(T x, T y) {
    if(x < y)
        return y;
    else
        return x;
}
```

How to use templates?

- Templates can define enum and typedef members.
- Templates can call templates
- Template parameters can be specialized

How to use templates?

- Templates can define enum and typedef members.
- Templates can call templates
- Template parameters can be specialized

How to use templates?

- Templates can define enum and typedef members.
- Templates can call templates
- Template parameters can be specialized

Factorial

Iterative implementation

```
int fac(int n) {  
    int x = 1;  
    for(int i=1;i<=n;i++) {  
        x *= i;  
    }  
    return x;  
}
```

Factorial

How to do that with templates?

- For loop is evaluated dynamically
- We want static evaluation
- Solution: recursive calls to templates will be evaluated statically

Factorial

How to do that with templates?

- For loop is evaluated dynamically
- We want static evaluation
- Solution: recursive calls to templates will be evaluated statically

Factorial

How to do that with templates?

- For loop is evaluated dynamically
- We want static evaluation
- Solution: recursive calls to templates will be evaluated statically

Factorial

Recursive implementation

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

Factorial

Template implementation

```
template<int N>
class Factorial {
public:
    enum {
        value = N * Factorial<N-1>::value
    };
};
```

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

Factorial

Template implementation

```
template<>
class Factorial<0> {
public:
    enum {value = 1 };
};

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

Factorial

Template implementation

```
template<int N>
class Factorial {
public:
    enum {
        value = N * Factorial<N-1>::value
    };
};
```

```
template<>
class Factorial<0> {
public:
    enum {value = 1 };
};
```

Factorial

Execution of the code

```
[michi@kopernic HS]$ g++ -o fact fact.cpp  
[michi@kopernic HS]$ ./fact  
5! ist 120  
7! ist 5040  
[michi@kopernic HS]$ █
```

Factorial

Inspection of the assembler code

```

.LCFI11:
    movl    %esp, %ebp
.LCFI12:
    pushl   %ecx
.LCFI13:
    subl   $36, %esp
.LCFI14:
    movl    $120, -8(%ebp)
    movl    $.LC0, 4(%esp)
    movl    $_ZSt4cout, (%esp)
    call   _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
    movl    %eax, %edx
    movl    -8(%ebp), %eax
    movl    %eax, 4(%esp)
    movl    %edx, (%esp)
    call   _ZNSolsEi
    movl    $_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_, 4(%esp)
    movl    %eax, (%esp)
    call   _ZNSolsEPFRSoS E
    movl    $5040, -8(%ebp)
    movl    $.LC1, 4(%esp)
    movl    $_ZSt4cout, (%esp)
    call   _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
    movl    %eax, %edx
    movl    -8(%ebp), %eax
    movl    %eax, 4(%esp)

```

-----XEmacs: fact.s (Assembler Font)-----L75--24%

Language constructs

We already used two constructs:

- Explicit specialization
- Recursion

Language constructs

We already used two constructs:

- Explicit specialization
- Recursion

Partial Specialization

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

```
template<int n>
struct power<n,0> {
    enum { RET = 1 };
};
```

Does not work with older compilers

Partial Specialization

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

```
template<int n>
struct power<n,0> {
    enum { RET = 1 };
};
```

Does not work with older compilers

Partial Specialization

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

```
template<int n>
struct power<n,0> {
    enum { RET = 1 };
};
```

Does not work with older compilers

IF-Construct

```
template<bool cond, class ThenType, class ElseType>
struct IF {
    typedef ThenType RET;
};
```

```
template<class ThenType, class ElseType>
struct IF<false, ThenType, ElseType> {
    typedef ElseType RET;
};
```

Can be done without partial specialization, as well. Similar constructs for while, do..loop, for and switch case can also be implemented using templates.

IF-Construct

```
template<bool cond, class ThenType, class ElseType>  
struct IF {  
    typedef ThenType RET;  
};
```

```
template<class ThenType, class ElseType>  
struct IF<false, ThenType, ElseType> {  
    typedef ElseType RET;  
};
```

Can be done without partial specialization, as well. Similar constructs for `while`, `do..loop`, `for` and `switch case` can also be implemented using templates.

IF-Construct

```
template<bool cond, class ThenType, class ElseType>  
struct IF {  
    typedef ThenType RET;  
};
```

```
template<class ThenType, class ElseType>  
struct IF<false, ThenType, ElseType> {  
    typedef ElseType RET;  
};
```

Can be done without partial specialization, as well. Similar constructs for `while`, `do..loop`, `for` and `switch case` can also be implemented using templates.

Partial evaluation

- Template functions can only evaluate static values
- Functions with at least one static parameter can at least be partially evaluated

```
template<int n>
inline int power(const int& m) {
return power<n-1>(m) * m;
}

template<>
inline int power<1>(const int& m) {
return m;
}
```

```
template<>
inline int power<0>(const int& m) {
return 1;
}
```

Partial evaluation

- Template functions can only evaluate static values
- Functions with at least one static parameter can at least be partially evaluated

```
template<int n>
inline int power(const int& m) {
return power<n-1>(m) * m;
}

template<>
inline int power<1>(const int& m) {
return m;
}
```

```
template<>
inline int power<0>(const int& m) {
return 1;
}
```

Template template parameters

Templates can take other templates as parameters

```
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 };
};
```

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

int void() {
    cout << Accumulate<3,Square>::RET << endl;
}
```

Template template parameters

Templates can take other templates as parameters

```
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 };
};
```

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

int void() {
    cout << Accumulate<3,Square>::RET << endl;
}
```

Template template parameters

Templates can take other templates as parameters

```
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 };
};
```

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

int void() {
    cout << Accumulate<3,Square>::RET << endl;
}
```

Loop unrolling

```
template< int i >
class LOOP{
public:
    static inline void EXEC(){
        cout << "A-" << i << " ";
        LOOP< i-1 >::EXEC();
        cout << "B-" << i << " ";
    }
};
```

```
template<>
class LOOP< 0 >{
public:
    static inline void EXEC(){
        cout << "A-0";
        cout << "\n";
        cout << "B-0 ";
    }
};
```

Output:

A-8 A-7 A-6 A-5 A-4 A-3 A-2 A-1 A-0

B-0 B-1 B-2 B-3 B-4 B-5 B-6 B-7 B-8

Loop unrolling

```
template< int i >
class LOOP{
public:
    static inline void EXEC(){
        cout << "A-" << i << " ";
        LOOP< i-1 >::EXEC();
        cout << "B-" << i << " ";
    }
};
```

```
template<>
class LOOP< 0 >{
public:
    static inline void EXEC(){
        cout << "A-0";
        cout << "\n";
        cout << "B-0 ";
    }
};
```

Output:

A-8 A-7 A-6 A-5 A-4 A-3 A-2 A-1 A-0

B-0 B-1 B-2 B-3 B-4 B-5 B-6 B-7 B-8

Loop unrolling

```
template< int i >
class LOOP{
public:
    static inline void EXEC() {
        cout << "A-" << i << " ";
        LOOP< i-1 >::EXEC();
        cout << "B-" << i << " ";
    }
};
```

Output:

A-8 A-7 A-6 A-5 A-4 A-3 A-2 A-1 A-0

B-0 B-1 B-2 B-3 B-4 B-5 B-6 B-7 B-8

```
template<>
class LOOP< 0 >{
public:
    static inline void EXEC() {
        cout << "A-0";
        cout << "\n";
        cout << "B-0 ";
    }
};
```

Functional elements in Template C++

- values can only be assigned once
- no loops
- pattern matching
- higher order functions

Functional elements in Template C++

- values can only be assigned once
- no loops
- pattern matching
- higher order functions

Functional elements in Template C++

- values can only be assigned once
- no loops
- pattern matching
- higher order functions

Functional elements in Template C++

- values can only be assigned once
- no loops
- pattern matching
- higher order functions

Loki library

- generic c++ library for design purposes
- created by Andrei Alexandrescu for his book *Modern C++ Design*
- combines template metaprogramming, generic programming and other techniques

Loki library

- generic c++ library for design purposes
- created by Andrei Alexandrescu for his book *Modern C++ Design*
- combines template metaprogramming, generic programming and other techniques

Loki library

- generic c++ library for design purposes
- created by Andrei Alexandrescu for his book *Modern C++ Design*
- combines template metaprogramming, generic programming and other techniques

Abstract Factory Pattern

- Interface for creating families of polymorphic objects
- Ensures that only members of one family will be used

Example

Application with different skins. Every skin has Buttons which implement `AbstractButton`, and WindowBorders which implement `AbstractBorder`. When using an Abstract Factory to initialize the skins, problems like having a `CrazyButton` at the same time with a `ClassicBorder` will not occur.

Abstract Factory Pattern

- Interface for creating families of polymorphic objects
- Ensures that only members of one family will be used

Example

Application with different skins. Every skin has Buttons which implement `AbstractButton`, and WindowBorders which implement `AbstractBorder`. When using an Abstract Factory to initialize the skins, problems like having a `CrazyButton` at the same time with a `ClassicBorder` will not occur.

Abstract Factory Pattern

- Interface for creating families of polymorphic objects
- Ensures that only members of one family will be used

Example

Application with different skins. Every skin has Buttons which implement `AbstractButton`, and WindowBorders which implement `AbstractBorder`. When using an Abstract Factory to initialize the skins, problems like having a `CrazyButton` at the same time with a `ClassicBorder` will not occur.

Abstract Factory implementation

- concrete Factory implementation for every specific family of objects
- implementation can be a lot of work
- all objects have to be in the factory
- Loki helps to use the Abstract Factory Pattern with a lot less work.

Abstract Factory implementation

- concrete Factory implementation for every specific family of objects
- implementation can be a lot of work
- all objects have to be in the factory
- Loki helps to use the Abstract Factory Pattern with a lot less work.

Abstract Factory implementation

- concrete Factory implementation for every specific family of objects
- implementation can be a lot of work
- all objects have to be in the factory
- Loki helps to use the Abstract Factory Pattern with a lot less work.

Abstract Factory implementation

- concrete Factory implementation for every specific family of objects
- implementation can be a lot of work
- all objects have to be in the factory
- Loki helps to use the Abstract Factory Pattern with a lot less work.

Abstract Factory implementation

- concrete Factory implementation for every specific family of objects
- implementation can be a lot of work
- all objects have to be in the factory
- Loki helps to use the Abstract Factory Pattern with a lot less work.

Abstract Factory in Loki

- Loki makes use of `Typelists` to implement the Abstract Factory Pattern
- `Typelists` are linked lists with types as values
- Good example of how to implement data types in template C++

Abstract Factory in Loki

- Loki makes use of `Typelists` to implement the Abstract Factory Pattern
- `Typelists` are linked lists with types as values
- Good example of how to implement data types in template C++

Abstract Factory in Loki

- Loki makes use of `Typelists` to implement the Abstract Factory Pattern
- `Typelists` are linked lists with types as values
- Good example of how to implement data types in template C++

Typelists

```
template <class T, class U>
struct Typelist {
    typedef T Head;
    typedef U Tail;
};
```

Usage: `typedef Typelist<char, Typelist<int, Typelist<double, NullType> > > Types;`

Typelists

```
template <class T, class U>  
struct Typelist {  
    typedef T Head;  
    typedef U Tail;  
};
```

Usage: `typedef Typelist<char, Typelist<int, Typelist<double, NullType> > > Types;`

Typelists in Loki

- Loki holds definitions to make typelists easier to use
- `typedef TYPELIST_3(char, int, double) Types;`
- Loki also has some functions for the typelist

Length function

```
template <class TList> struct Length;
template <> struct Length<NullType> {
    enum {value = 0};
};
template <class T, class U>
struct Length<TypeList<T, U> > {
    enum { value = 1 + Length<U>::value };
};
```

Sophisticated Typelist functions

- Loki provides typelist functions to create classes.
- These functions will apply all types in the typelist to a template.

Sophisticated Typelist functions

- Loki provides typelist functions to create classes.
- These functions will apply all types in the typelist to a template.

Sophisticated Typelist functions

- Loki provides typelist functions to create classes.
- These functions will apply all types in the typelist to a template.

```
template<class T>  
struct Holder {  
    T value_;  
};
```

Abstract Factory instantiation

- ```
typedef AbstractFactory<
 TYPELIST_2(Button, ScrollBar)
> AbstractAppFactory;
```
- To create a specialized version of this factory Loki uses a template type called policy.
- Policies are small template classes which define a certain behaviour, in this case the creation of the factories objects
- ```
typedef ConcreteFactory<
    AbstractAppFactory,
    OpNewFactoryUnit,
    TYPELIST_2(ModernButton, ModernScrollBar)
> ModernAppFactory;
```
- Loki provides all templates for the Abstract Factory pattern.

Abstract Factory instantiation

- ```
typedef AbstractFactory<
 TYPELIST_2(Button, ScrollBar)
> AbstractAppFactory;
```
- To create a specialized version of this factory Loki uses a template type called policy.
- Policies are small template classes which define a certain behaviour, in this case the creation of the factories objects

- ```
typedef ConcreteFactory<
    AbstractAppFactory,
    OpNewFactoryUnit,
    TYPELIST_2(ModernButton, ModernScrollBar)
> ModernAppFactory;
```

- Loki provides all templates for the Abstract Factory pattern.

Abstract Factory instantiation

- ```
typedef AbstractFactory<
 TYPELIST_2(Button, ScrollBar)
> AbstractAppFactory;
```
- To create a specialized version of this factory Loki uses a template type called policy.
- Policies are small template classes which define a certain behaviour, in this case the creation of the factories objects
- ```
typedef ConcreteFactory<
    AbstractAppFactory,
    OpNewFactoryUnit,
    TYPELIST_2(ModernButton, ModernScrollBar)
> ModernAppFactory;
```
- Loki provides all templates for the Abstract Factory pattern.

Abstract Factory summary

- Instead of writing an Abstract Factory class and its concrete implementations, library calls with a few short lines of code.
- Can be further optimized using the Prototype pattern.
- Loki implements more design patterns in this manner using sophisticated template metaprogramming.

Abstract Factory summary

- Instead of writing an Abstract Factory class and its concrete implementations, library calls with a few short lines of code.
- Can be further optimized using the Prototype pattern.
- Loki implements more design patterns in this manner using sophisticated template metaprogramming.

Abstract Factory summary

- Instead of writing an Abstract Factory class and its concrete implementations, library calls with a few short lines of code.
- Can be further optimized using the Prototype pattern.
- Loki implements more design patterns in this manner using sophisticated template metaprogramming.

Template C++ in libraries

- BoostML
- Blitz++
- Pooma
- Spirit
- MTL

BoostML is a general-purpose, high-level library was created by Aleksey Gurtovoy and David Abrahams in 2002 and is discussed in their book C++ Template Metaprogramming.

Template C++ in libraries

- BoostML
- Blitz++
- Pooma
- Spirit
- MTL

Blitz++ is a library for scientific computing. Its performance is on par with Fortran 77/90. Todd Veldhuizen created the core of Blitz++, by now several people are contributing to it.

Template C++ in libraries

- BoostML
- Blitz++
- Pooma
- Spirit
- MTL

Pooma was developed at Los Alamos National Laboratory as a general purpose library for parallel scientific computations. It uses expression templates to achieve a performance comparable to Fortran 90.

Template C++ in libraries

- BoostML
- Blitz++
- Pooma
- Spirit
- MTL

Spirit is a object-oriented recursive-decent parser generator library made by Joel de Guzman in 1998.

Template C++ in libraries

- BoostML
- Blitz++
- Pooma
- Spirit
- MTL

The Matrix Template Library provides high-performance linear algebra functionality for a wide variety of matrix formats.

Disadvantages

- Computes only static values
- Little portability
- Debugging issues
- Readability
- Compiling takes longer

Disadvantages

- Computes only static values
- Little portability
- Debugging issues
- Readability
- Compiling takes longer

Disadvantages

- Computes only static values
- Little portability
- Debugging issues
- Readability
- Compiling takes longer

Disadvantages

- Computes only static values
- Little portability
- Debugging issues
- Readability
- Compiling takes longer

Disadvantages

- Computes only static values
- Little portability
- Debugging issues
- Readability
- Compiling takes longer

Advantages

- + code is evaluated at compile-time
- + can obtain and use metainformation
- + can create dynamic code

Advantages

- + code is evaluated at compile-time
- + can obtain and use metainformation
- + can create dynamic code

Advantages

- + code is evaluated at compile-time
- + can obtain and use metainformation
- + can create dynamic code

Summary

- Compile-time evaluated code which can manipulate and create dynamic code
- Great for design or mathematical libraries
- Drawbacks make it hard to use in normal programs

Summary

- Compile-time evaluated code which can manipulate and create dynamic code
- Great for design or mathematical libraries
- Drawbacks make it hard to use in normal programs

Summary

- Compile-time evaluated code which can manipulate and create dynamic code
- Great for design or mathematical libraries
- Drawbacks make it hard to use in normal programs