C++ Template Metaprogramming

Juan Pedro Bolívar Puente

Ableton AG

16 May 2013

Agenda

1. Introduction
2. Concepts and techniques
3. Reasoning about metaprograms
4. Conclusions

Agenda

What is metaprogramming?

Metaprogramming is writing programs that <u>generate</u> or <u>manipulate</u> code

Metaprogramming with C preprocessor

Back in the old C days...

```
#define DECLARE_LIST_OF(TYPE)          \
struct node_of_ ##TYPE {               \
    TYPE data;                         \
    struct node_of_ ##TYPE * next;     \
};

DECLARE_LIST_OF(int);
```

Metaprogramming with C preprocessor

...and not so old, take look at Boost.PP...

```
#define DECLARE_LIST_OF(TYPE)         \
struct node_of_ ##TYPE {              \
    TYPE data;                        \
    struct node_of_ ##TYPE * next;    \
};

DECLARE_LIST_OF(int);
```

Templates in C++

...in the beginning, templates were **macros that know** a bit about types and syntax...

```
template<typename T>
class vector
{
    T* _data;
    typedef T& reference_type;
    reference_type operator[] (size_t index);
    ...
};
```

Templates and specialization

...and then things got crazy...

```
template<typename T>
class vector { ... };

template<typename U>
class vector<U*> : private vector<void*>
{ ... };

template<>
class vector<bool> { ... };
```

This shit is <u>Turing complete</u>!

Agenda

My first non-trivial metaprogram

```
template<long N> struct fib_c
{
  static const long value =
       fib_c<N-1>::type + fib_c<N-2>::type;
  typedef fib_c type;
};

template<> struct fib_c<0> {
  static const long value = 0; ...
};

template<> struct fib_c<1> {
  static const long value = 1; ...
};
```

My first non-trivial metaprogram

```
template<long N> struct fib_c
{
  static const long value =
      fib_c<N-1>::type + fib_c<N-2>::type;
  typedef fib_c type;
};

template<> struct fib_c<0> {
  static const long value = 0; ...
};

template<> struct fib_c<1> {
  static const long value = 1; ...
};
```

My first non-trivial metaprogram

```
template<long N> struct fib_c
{
  static const long value =
        fib_c<N-1>::type + fib_c<N-2>::type;
  typedef fib_c type;
};

template<> struct fib_c<0> {
  static const long value = 0; ...
};

template<> struct fib_c<1> {
  static const long value = 1; ...
};
```

My first non-trivial metaprogram

```
template<long N> struct fib_c
{
  static const long value =
       fib_c<N-1>::type + fib_c<N-2>::type;
  typedef fib_c type;
};

template<> struct fib_c<0> {
  static const long value = 0; ...
};

template<> struct fib_c<1> {
  static const long value = 1; ...
};
```

My first non-trivial metaprogram

```
template<long N> struct fib_c
{
  static const long value =
      fib_c<N-1>::type + fib_c<N-2>::type;
  typedef fib_c type;
};

template<> struct fib_c<0> {
  static const long value = 0; ...
};

template<> struct fib_c<1> {
  static const long value = 1; ...
};
```

My first non-trivial metaprogram

```
#include <iostream>
int main()
{
    std::cout << fib_c<90>::value
              << std::endl;
    return 0;
}
```

This program runs in constant time!

My first non-trivial metaprogram

```
template<long N> struct fib_c
{
   static const long value =
        fib_c<N-1>::type + fib_c<N-2>::type;
   typedef fib_c type;
};

template<> struct fib_c<0> {
   static const long value = 0; ...
};

template<> struct fib_c<1> {
   static const long value = 1; ...
};
```

For high values of *N* this metafunction is much faster than equivalent runtime C++ code, **why**?

The templates metatype-system

1. Types
2. Integrals
3. Meta-functions

The templates metatype-system

1. Types

```
template<typename T>
struct add_pointer
{ typedef T* type; }

template<class T>
struct identity
{ typedef T type; }
```

2. Integrals
3. Meta-functions

The templates metatype-system

1. Types
2. Integrals

```
template<long> fib_c;
```

3. Meta-functions

The templates metatype-system

1. Types
2. Integrals
3. Meta-functions

```
template<template<class> F, class V, int N>
struct repeat_c {
  typedef typename repeat_c<typename F<V>::type,
                            N-1>::type type;
};

template<template<class> F, class V>
struct repeat_c<F, V, 0> : identity<V> {};
```

The templates metatype-system

1. Types
2. ~~Integrals~~
3. ~~Meta-functions~~

Lets become untyped via **type erasure**

The Metafunction concept

- A *N*-ary metafunction is a template over *N* type parameters types that has a nested **type**

  It is **partial** if defines **type** for a some specializations only
- A nullary metafunction is a concrete type with a nested **type** concrete type.

```
auto concept Metafunction<F> {
    typename type = F::type;
};
```

Erasing integral metatypes...

We box integrals into a **nullary metafunction**

```
template<typename T, T V>
struct integral_c {
    static const T value = V;
    typedef integral_c type;
};

template<bool V> struct bool_c
    : integral_c<bool, V> {};

typedef bool_c<true> true_;
typedef bool_c<false> false_;
```

Erasing integral metatypes...

See the Boost.MPL **IntegralConstant** concept

```
template<typename T, T V>
struct integral_c {
    static const T value = V;
    typedef integral_c type;
};

template<bool V> struct bool_c
    : integral_c<bool, V> {};

typedef bool_c<true> true_;
typedef bool_c<false> false_;
```

Lazy evaluation

A metafunction is lazy if its arguments **can** be passed unevaluated.

```
template<bool C, typename T1, typename T2>
struct if_c { typedef T1 type; };
template<typename T1, typename T2>
struct if_c<false, T1, T2> { typedef T2 type; };

template<typename C, typename T1, typename T2>
struct if_ : if_c<C::value, T1, T2> {}

template<typename P, typename T1, typename T2>
struct eval_if :
    if_c<P::type::value, T1, T2>::type {}
```

Lazy evaluation

Beware that eval_if in MPL
dœs not evaluate the predicate

```
template<bool C, typename T1, typename T2>
struct if_c { typedef T1 type; };
template<typename T1, typename T2>
struct if_c<false, T1, T2> { typedef T2 type; };

template<typename C, typename T1, typename T2>
struct if_ : if_c<C::value, T1, T2> {}

template<typename P, typename T1, typename T2>
struct eval_if :
    if_c<P::type::value, T1, T2>::type {}
```

Fibonnaci numbers revisited

fib is now a model of metafunction

```
using namespace boost::mpl;

template<typename N>
struct fib :
    eval_if<equal_to<N, long_<0> >, long_<0>,
    eval_if<equal_to<N, long_<1> >, long_<1>,
            plus<fib<minus<N, long_<1> > >,
                 fib<minus<N, long_<2> > > > > >
        ::type {};
```

Fibonnaci numbers revisited

Being too lazy broke **memoization**!

```
using namespace boost::mpl;

template<typename N>
struct fib :
  eval_if<equal_to<N, long_<0> >, long_<0>,
  eval_if<equal_to<N, long_<1> >, long_<1>,
    plus<fib<minus<typename N::type, long_<1> > >,
         fib<minus<typename N::type, long_<2> > > >
    > >::type {};
```

Erasing metafunction metatypes

A metafunction class is a type with
a nested **apply** metafunction

```
struct fib_f {
    template<typename N>
    struct apply
        : fib<N> {};
    typename fib_f type;
};
```

Erasing metafunction metatypes

A metafunction class is a type with
a nested **apply** metafunction

```
using namespace boost::mpl;

typedef
  lambda<fib<_1> >::type
  fib_f;
```

Erasing metafunction metatypes

A metafunction class is a type with
a nested **apply** metafunction

```
using namespace boost::mpl;

typedef
  lambda<fib<_1> >::type
  fib_f;
```

Agenda

1. Introduction
2. Concepts and techniques
3. Reasoning about metaprograms
4. Conclusions

The metaprogramming language

C++ templates as a language are...

- Purely functional
- Lazyly evaluated
- Untyped[1]

[1] See *tag dispatching* in MPL to see how to add type classes

The metaprogramming language

C++ templates as a language are...

- Purely functional like Haskell
- Lazyly evaluated like Haskell
- ~~Untyped~~[1]

[1] See *tag dispatching* in MPL to see how to add type classes

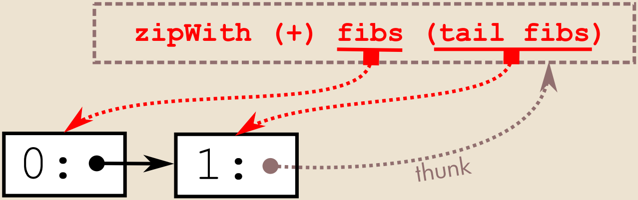
The Haskell Fibonacci sequence

We can translate many Haskell programs into C++ metaprograms

```
fibs = 0 : 1 :
       zipWith (+) fibs (tail fibs)

fib n = fibs !! n
```

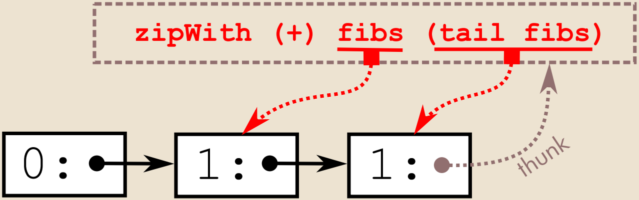
The Haskell Fibonacci sequence




```
fibs = 0 : 1 :
        zipWith (+) fibs (tail fibs)

fib n = fibs !! n
```

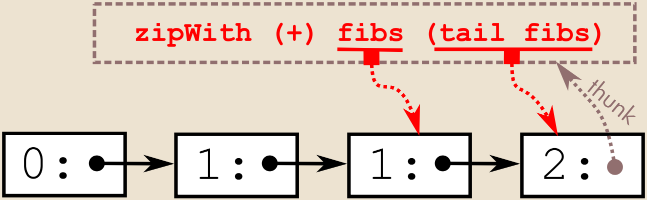
The Haskell Fibonacci sequence




```
fibs = 0 : 1 :
       zipWith (+) fibs (tail fibs)

fib n = fibs !! n
```

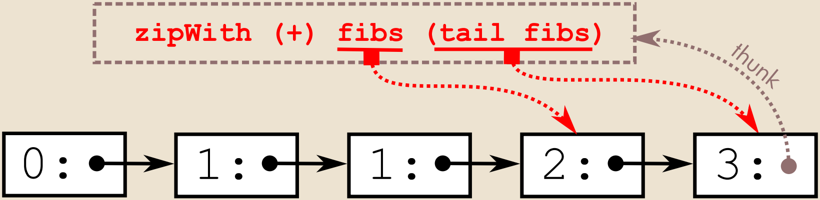
The Haskell Fibonacci sequence




```
fibs = 0 : 1 :
       zipWith (+) fibs (tail fibs)

fib n = fibs !! n
```

The Haskell Fibonacci sequence




```haskell
fibs = 0 : 1 :
       zipWith (+) fibs (tail fibs)

fib n = fibs !! n
```

Functional lists

Lets add a **list cell** and trivial metafunctions

```
struct void_ { typedef void_ type; };

template<typename Head, typename Tail = void_>
struct cons {
  typedef Head head;
  typedef Tail tail;
  typedef cons type;
};

template<typename T> struct head
{ typedef typename T::head type; };
...
```

High order list functions

```
zipWith f (x1:xs1) (x2:xs2) =
     f x1 x2 : zipWith xs1 xs2
zipWith f [] _ = []
zipWith f _ [] = []
```

Note how we use structural recursion
instead of tail recursion

Haskell zipWith in templates

```
template<typename F, typename List1,typename List2>
struct zip_with {
    typedef typename List1::type l1;
    typedef typename List2::type l2;
    typedef typename L1::head    x1;
    typedef typename L2::head    x2;
    typedef typename L1::tail    xs1;
    typedef typename L2::tail    xs2;
    typedef typename
    eval_if<or_<is_same<l1, void_>,
                is_same<l2, void_> >,
            void_,
            cons<typename apply<F, x1, x2>::type,
                 zip_with<F, xs1, xs2> > >
      ::type type;
};
```

List random access

```haskell
(x:xs) !! n = xs !! (n-1)
(x:_)  !! 0 = x
```

```cpp
template<typename Num, typename List>
struct at_ {
    typedef typename List::type L;
    typedef typename Num::type  N;
    typedef typename
    eval_if<equal_to<N, long_<0> >,
            head<L>,
            at_<prior<N>, tail<L> > >
      ::type type;
};
```

Fibonacci infinite list

```
struct fibs :
  cons<long_<0>,
  cons<long_<1>,
       zip_with<plus<_1, _2>,
                fibs,
                tail<fibs> > > > {};

template<typename N>
struct fib : at_<N, fibs> {};
```

Agenda

1. Introduction
2. Concepts and techniques
3. Reasoning about metaprograms
4. Conclusions

How expensive is this?

Use **gcc -ftime-report** to see the template instantiation cost

N	fib0	fib1-1	fib1-2	fib2	fib-dyn
5	0.01s	0.03s	0.05s	0.04s	0.00s
10	0.02s	0.14s	0.06s	0.05s	0.00s
15	0.02s	1.14s	0.06s	0.08s	0.00s
20	0.03s	14.40s	0.07s	0.09s	0.00s
30	0.03s		0.08s	0.11s	0.01s
50	0.04s		0.10s	0.16s	
70	0.04s		0.12s	0.20s	
90	0.04s		0.15s	0.26s	

How expensive is this?

Use **gcc -ftime-report** to see the template instantiation cost

N	fib0	fib1-1	fib1-2	fib2	fib-dyn
5	3.3MB	4.4MB	4.1MB	5.3MB	>1MB
10	3.3MB	13.2MB	4.7MB	6.7MB	>1MB
15	3.4MB	110.4MB	5.4MB	8.1MB	>1MB
20	3.4MB	1.2GB	6.0MB	9.5MB	>1MB
30	3.4MB		7.3MB	12.2MB	>1MB
50	3.5MB		9.8MB	17.9MB	
70	3.6MB		12.4MB	23.3MB	
90	3.7MB		15.5MB	28.7MB	

The good side of it

- It teaches us functional programming
- Helps understanding Boost compilation errors
- Useful for generic yet efficient and safe code

The ugly side of it

- Until C++11, hacks are required

  Template aliases, variadic templates, constexpr, perfect forwarding...
- Compilation errors are just the evaluation tree

  Until we get **concepts**, someday...
- Can harm compilation time
- Ugly syntax

Further readings

- C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond
  David Abrahams, Aleksey Gurtovoy
  Addison-Wesley Professional, 2001

- Modern C++ Design: Generic Programming and Design Patterns Applied
  Andrei Alexandrescu
  Addison-Wesley Professional, 2001

Questions?

Thanks for listening!