# Static Metaprogramming in C++

Haoyuan Li

Dipartimento di Informatica

Università di Pisa


Università di Pisa

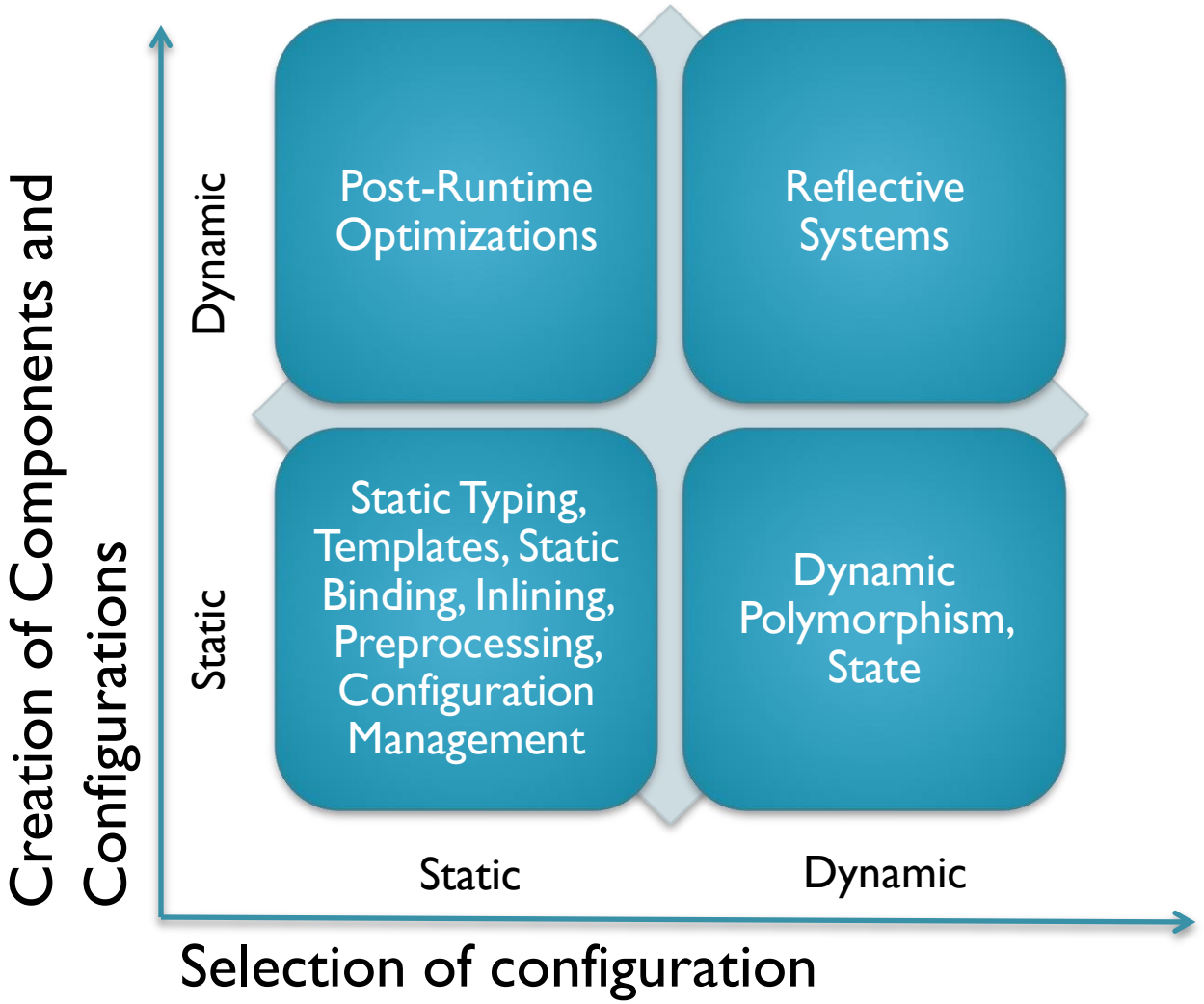
# Generic versus Generative

- Generic programming focuses on representing families of domain concepts (*parameterization*)
- Generative programming also includes the process of creating concrete instances of concepts (*metaprogramming*)
- In generative programming, the principles of generic programming are applied to the solution space

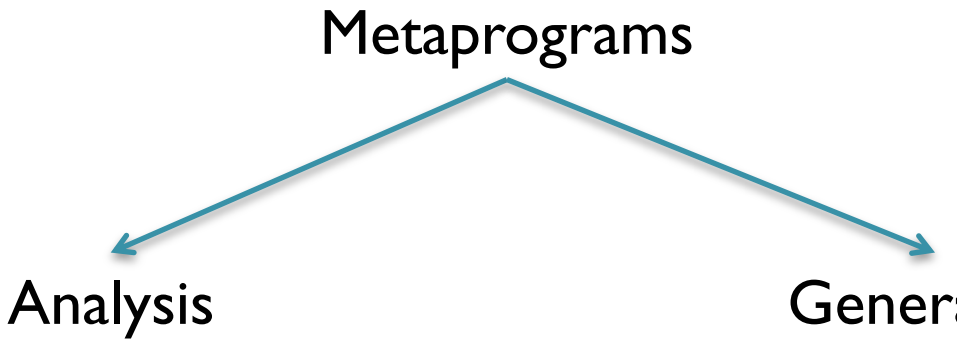
# Metaprogramming

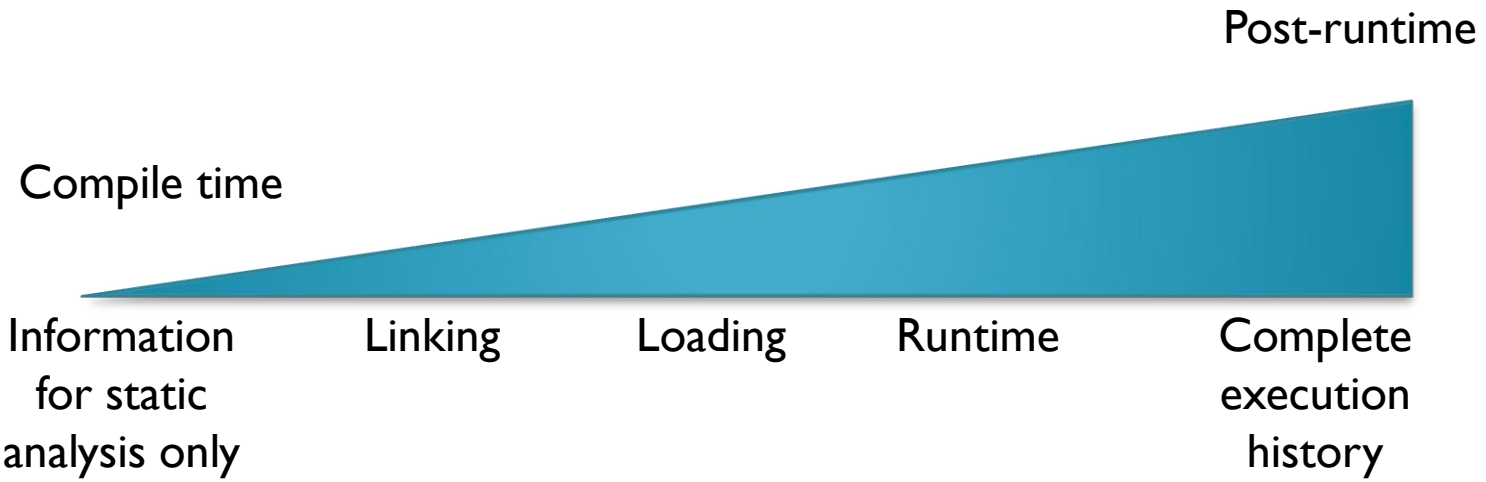
- Automating the production of individually configured systems
- Building adaptive systems that need to be able to dynamically adjust themselves to a changing environment
- Writing programs that represent and manipulate other programs or themselves (reflection)

# Types of System Configuration



# Metaprograms



# C++ Support for Configuration

- Dynamic Configuration
  - Dynamic polymorphism (virtual methods)
- Static Configuration
  - Static typing
  - Static binding
  - Inlining
  - Templates
  - Parameterized inheritance
  - **typedef**
  - Member types and nested classes

# C++ as a Two-Level Language

- Dynamic code is compiled and executed at runtime
- Static code is evaluated at compile time
- C++ template together with a number of other C++ features constitute a Turing-complete compile-time language!
  - Looping – template recursion
  - Conditional – template specialization

# Example of C++ Static Code

```
template<int i>
class C {
    enum { RET = i };
};

cout << C<2>::RET << endl; // Output 2

int n = 2;
cout << C<n>::RET << endl; // ???
```

# Dynamic Factorial Computing

```
int factorial(int n)
{
    return (n == 0) ? 1 : n * factorial(n – 1);
}

cout << factorial(7) << endl; // 5040
```

# Static Factorial Computing

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

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

cout << Fact<7>::RET << endl; // 5040

int i = 3;
Fact<i>::RET; // ???
```

# Dynamic Fibonacci Number

```
int fibo(long long int n)
{
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    return fibo(n - 1) + fibo(n - 2);
}

cout << fibo(45) << endl;

// 1134903170
// 31.890s with Intel Core 2 Duo 2.4GHz CPU!
```

# Static Fibonacci Number

```
template<long long int n>
struct Fibo {
    static const long long int RET =
        Fibo<n-1>::RET + Fibo<n-2>::RET;
};

template<>
struct Fibo<0> {
    static const long long int RET = 0;
};

template<>
struct Fibo<1> {
    static const long long int RET = 1;
};
```

# Efficiency of Static Code

```
long long int x = Fibo<45>::RET;
cout << x << endl;

// 1134903170
// runtime 0.006s
// compile time 0.314s

long long int x = Fibo<500>::RET; // !!!!!
cout << x << endl;

// 2171430676560690477
// runtime 0.005s
// compile time 0.373s
```

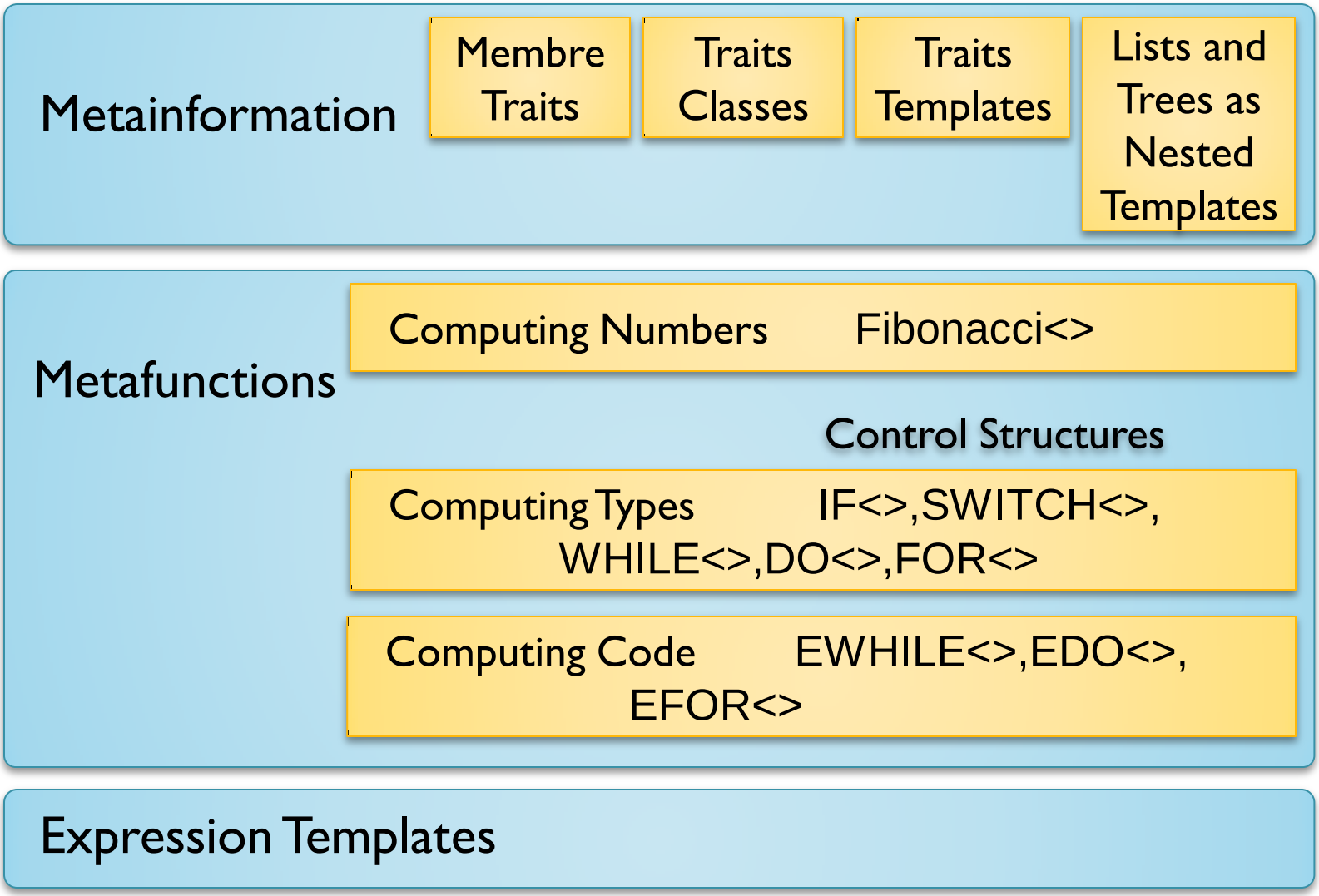
# Why?

- There is no double-recursion in static code!
- Compiler sees **Fib<7>::RET**, it instantiates the structure template **Fib<>** for **n=7**
- At this point, the compiler also has to instantiate **Fib<6>::RET** and **Fib<5>::RET**
- And then for **n=4**, **n=3**, **n=2**, **n=1**, and **n=0**
- We can regard **Fib<>** as a function, which is evaluated at compile time
- The dynamic code only does "<<"!

# Functional Flavor of Static Level

- Class templates as functions
- Integers and types as data
- Symbolic names instead of variables
- Constant initialization and **typedef** instead of assignment
- Template recursion instead of loops
- Conditional operator and template specialization as conditional construct

# Template Metaprogramming Map



# Member Traits

```
struct Car {
    enum { car, alfaromeo, fiat, peugeot };
    enum { id = car};
};
struct AlfaRomeo: Car {
    enum { id = alfaromeo };
};
struct Fiat: Car {
    enum { id = fiat };
};
struct Peugeot: Car {
    enum { id = peugeot };
};

myCar.id == myCar::peugeot; // runtime type test
```

# Traits Templates

```
template<class T>
class car_traits {
public:
    enum { MADE, ITALY, FRANCE };
    enum { FUEL, DIESEL, PETROL };
    static const int made = MADE;
    static const int fuel = FUEL;
    static const bool trans = 0;
    static const char gearbox = 0;
    static const char ports = 0;
    static const double volume = 0.0;
};
```

# Traits Templates: Specialization

```
template<>
class car_traits<Peugeot> {
public:
    enum { MADE, ITALY, FRANCE };
    enum { FUEL, DIESEL, PETROL };
    static const char made = FRANCE;
    static const char fuel = FUEL;
    static const char ports = 0;
    static const char speeds = 0;
    static const double volume = 0.0;
};
```

# Traits Templates: Extension

```
class DieselCar {
public:
    enum { FUEL, DIESEL, PETROL };
    static const int fuel = DIESEL;
};

template<>
class car_traits<MyCar>: public DieselCar {
    static const char ports = 5;
    static const char speeds = 5;
    static const double volume = 1.9;
};
```

# IF<> with Partial Specialization

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

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

IF<(1+2>4), short, int>::RET i; // i is int
```

# Usage of IF<> Structure

```
class DieselMotor: Motor {
public:
    virtual void assemble() {}
};

class PetrolMotor: Motor {
public:
    virtual void assemble() {}
};

IF<Car::motor == Car::DIESEL,
   DieselMotor, PetrolMotor>::RET motor;

motor.assemble();
```

# Other Operators

```
struct FalseType {};
struct TrueType {};

template <class T>
struct isPointer { typedef FalseType RET; };

template <class T>
struct isPointer<T*> {
   typedef TrueType RET; };
```

# Conditional

```
template<int n1, int n2>
struct Max {
   enum { RET = (n1 > n2) ? n1 : n2 };
}
```

# Template Metafunctions

C(k, n) = n! / (k! * (n-k)!)

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

cout << Comb<2, 4>::RET << endl;
```

# Functions on Templates

```
template <class L>
struct Length {
   enum { RET = 1 + Length<L::next>::RET
   };
};
template <>
struct Length<Nil> { enum RET = 0 };
```

# Possible Uses

- Control loop unrolling:

  FOR<3, B>::loop;

# Representing Data Structures

```
template<int n, class Next>
struct List {
    enum { item = n };
    typedef Next next;
};

struct Nil { };

typedef List<1, List<2, Nil()> > Vector;

V::item == 1;
V::next::item == 2;
V::next::next == Nil;
```

# Length of a List

```
template<class L>
struct Length {
    enum { RET = 1 + Length<L::next>::RET };
};

template<>
struct Length<Nil> { enum RET = 0 };

cout << Length<Vector> << endl;

// What's the length of Vector?
```

# Unrolling Vector Addition

```
void add(size_t size, int* a, int* b, int* c)
{
    while (size--)
        *c++ = *a++ + *b++;
}
```

# Template Unrolling

```
template<int size>
inline void add(int* a, int* b, int* c)
{
    *c = *a + *b;
    add<size-1>(a + 1, b + 1, c + 1);
}

template<>
inline void add<0>(int* a, int* b, int* c) { }

add<3>(a, b, c);
```

# Effect of Unrolling

```
add<3>(a, b, c);
    *c = *a + *b;
    add<2>(a + 1, b + 1, c + 1);
        *(c + 1) = *(a + 1) + *(b + 1);
        add<1>(a + 2, b + 2, c + 2);
            *(c + 2) = *(a + 2) + *(b + 2);
            add<0>(a + 3, b + 3, c + 3);
```

# For-Loop

```
template<int n, class B>
struct FOR {
    static void loop(int n) {
        UNROLL<n, B>::iteration(0);
    }
};
```

# Loop Unrolling

```
template <int n, class B>
struct UNROLL {
    static void iteration(int i) {
        B::body(i);
        UNROLL<n-1, B>::iteration(i + 1);
    }
};

template <class B>
struct UNROLL<0, B> {
        static void iteration(int i) { }
};
```

# Loop Unrolling (2)

```
template <int n, class B>
struct UNROLL {
   static void iteration(int i) {
         B::body(i);
         UNROLL<n-1, B>::iteration(i + 1);
   }
};
template <class B>
struct UNROLL<0, B> {
   static void iteration(int i) { }
};
```

# Uses

- Blitz++
- Boost

# Blitz++

- A library for algebraic numeric computing
- Heavy use of template metaprogramming
- Allows achieving faster speed than dedicated Fortan numeric libraries

# Boost

- General C++ Class Library
- Many Uses of template metaprogrammming:
  - foreach construct, uses TMP to determie types of elements on which to iterate, distinguishes between constant and normal iterator types
  - Ublast: library of algebraic linear algebra (vector, matrix) uses TMP to optimize code for compound algebraic operations

# Problems

- Readability
- Debugging
- Error reporting
- Compilation time
- Compiler limits
- Expression templates limits
- Portability