C++模板元编程技术与应用

荣耀
royal@royaloo.com
www.royaloo.com

动机

让更多的C++程序员了解模板元编程，并在此过程中获得快乐！

目录

* 历史
* 导入范例
* 主要思想
* 静态语言设施
* 控制结构
* 数据结构
* 数值计算
* 类型计算
* 代码生成
* 断言和契约
* 库
* DSEL设计
* 结语
* 资源

历史

1994年，在圣迭哥举行的一次C++标准委员会会议期间，Erwin Unruh展示了一段特别的代码，可以在编译期以编译错误信息的方式产生从2到某个给定值之间的所有质数。

这份代码的原始版本见[注5]，修订版见[注6]。可以使用GCC编译器观察到上述效果。

同年夏天，Todd Veldhuizen受Erwin的例子启发，发现可以使用C++模板进行元编程（metaprogramming），并发表了一份技术报告。次年5月又在*C++ Report*上发表了一篇名为"*Using C++ template metaprograms*"的文章，从而将Erwin Unruh发现的C++编译期模板编程（Compile-time Template Programming）进一步精化为C++模板元编程（Template Metaprogramming，TMP）。

导入范例

1.计算Fibonacci数列第N项

```
// 主模板
template<int N>
struct Fib
{
    enum { Result = Fib<N-1>::Result + Fib<N-2>::Result };
};
```

主模板用于处理一般的逻辑

特化必须放在主模板之后

```
// 完全特化版
template <>
struct Fib<1>
{
    enum { Result = 1 };
};
```

处理N=1的情况

```
// 完全特化版
template <>
struct Fib<0>
{
    enum { Result = 0 };
};
```

处理N=0的情况

```
// 示例
int main()
{
    int i = Fib<10>::Result;
    // std::cout << i << std::endl;
}
```

导入范例

运作机理：当编译器实例化Fib<10>时，为了给其enum Result赋值，编译器需要对Fib<9>和Fib<8>进行实例化，同理，又需要针对Fib<7>和Fib<6>实例化同样的模板……，当实例化到Fib<1>和Fib<0>的时候，完全特化版被实例化，至此递归结束。这个处理过程类似于递归函数调用，编译器被用于解释元程序，生成的结果程序中仅包含一个常量值。

语句

```
int i = Fib<10>::Result;
```

被VC7.1编译成如下指令（Intel P4 CPU）：

```
00411A1E  mov         dword ptr [i],37h
```

字面量37h即为Fibonacci数列的第10项的值。可见，Fib<10>::Result的确被评估于编译期，结果作为处理器指令的一部分而被存储起来。

导入范例

2.类型选择

```
// 主模板
template<bool condition, typename T1, typename T2>
struct IfThenElse
{
    typedef T1 ResultType;
};

// 局部特化
template<typename T1, typename T2>
struct IfThenElse<false, T1, T2>
{
    typedef T2 ResultType;
};

// 示例
IfThenElse<(1 + 1 == 2), int, char>::ResultType i; // i的类型为int
```

基于给定的布尔常量表达式在两个类型之中二选一。若表达式为true，则T1被typedef为ResultT，否则ResultT代表T2。

只针对模板参数的局部进行特化。

主要思想

利用模板特化机制实现编译期条件选择结构，利用递归模板实现编译期循环结构，模板元程序则由编译器在编译期解释执行。

静态语言设施

模板元编程使用静态C++语言成分，编程风格类似于函数式编程，其中不可以使用变量、赋值语句和迭代结构等。

在模板元编程中，主要操作整型（包括布尔类型、字符类型、整数类型）常量和类型。被操纵的实体也称为元数据（Metadata）。所有元数据均可作为模板参数。

其他元数据类型还包括枚举、函数指针/引用、全局对象的指针/引用以及指向成员的指针等。另外，已经有一些编译期浮点数计算探索（参见[注7]）。

由于在模板元编程中不可以使用变量，我们只能使用typedef名字和整型常量。它们分别采用一个类型和整数值进行初始化，之后不能再赋予新的类型或数值。如果需要新的类型或数值，必须引入新的typedef名字或常量。

静态语言设施

编译期赋值通过整型常量初始化和typedef语句实现。例如：

```
enum { Result = Fib<N-1>::Result + Fib<N-2>::Result};
```

新、旧编译器均支持

或

```
static const int Result = Fib<N-1>::Result + Fib<N-2>::Result;
```

成员类型则通过typedef引入，例如：

```
typedef T1 Result;
```

静态语言设施

条件结构采用模板特化或条件操作符实现。如果需要从两个或更多种类型中选其一，可以使用模板特化，如前述的IfThenElse。可以使用条件操作符返回两个候选数值之一，例如：

```
template<int a, int b>
struct Max
{
    enum { Result = (a > b) ? a : b };
};
```

C++静态语言设施是图灵完备的，理论上可以用于实现任何可实现的算法。

静态C++代码使用递归而不是循环语句。递归的终结采用模板特化实现。如果没有充当终结条件的特化版，编译器将一直实例化下去，一直到达编译器的极限。

C++标准建议编译器实现至少要支持17层实例化。大多数编译器支持的递归实例化数目远不止17。例如，GCC支持多达500层递归模板实例化。

控制结构

可以使用模板元编程实现与运行期C++所对应的程序流程控制结构。

```
// If
// 主模板
template<bool>
struct If;

// 完全特化版
template<>
struct If<true>
{
    static void F()
    {
        // 待执行的语句
    }
};
```

主模板未必一定要予以定义。此主模板纯粹供随后的特化所用。

```
// 完全特化版
template<>
struct If<false>
{
    static void F()
    {
        // 待执行的语句
    }
};
```

```
// 示例
If<Condition>::F();
```

控制结构

```
// For
// 主模板
template<int N>
struct For
{
    static inline void f()
    {
        // 待执行的语句
        For<N-1>::f();
    }
};
```

```
// 完全特化版
template<>
struct For<0>
{
    static inline void f()
    {
      // 空，什么都不做
    }
};
```

我们必须给出这个什么也不做的f()，否则会导致N=1时实例化失败。

```
// 使用
For<10>::f();
```

类似地，可以给出While、Do-While实现

控制结构

```
// Switch
// 主模板
template<int defalutvalue>
struct Switch
{
    static void f()
    {
        // 缺省情况下执行的语句
    }
};
```

```
// 完全特化版1
template<>
struct Switch<value1>
{
    static void f()
    {
        // 待执行的语句1
    }
};
```

```
// 完全特化版2
template<>
struct Switch<value2>
{
    static void f()
    {
        // 待执行的语句2
    }
};
```

```
// 示例
Switch<valuex>::f();
```

该实现不够直观，在语法上和运行期switch相去甚远。一个更自然的实现应该支持类似于运行期switch的语法……

控制结构

```
struct A
{
    static void execute()
    {
        cout << "A" << endl;
    }
};

struct B
{
    static void execute()
    {
        cout << "B" << endl;
    }
};
```

```
struct C
{
    static void execute()
    {
        cout << "Default" << endl;
    }
};
```

我们希望支持这样的用法

```
// 用法示例
Switch<(1+1-2),
    Case<1, A,
    Case<2, B,
    Case<DEFAULT, C> > >
>::Result::execute(); // 打印"Default"
```

控制结构

```
// 主模板
template <int tag, typename Case>
struct Switch
{
private:
    typedef typename Case::Next NextCase;
    enum { caseTag = Case::tag,
        found = (caseTag == tag || caseTag ==
DEFAULT)};
public:
    typedef typename IfThenElse<found, typename
Case::Type, typename Switch<tag,
NextCase>::Result>::ResultType Result;
};
```

利用typename消除歧义

如前述

```
// Switch
const int DEFAULT = -1;

struct NilCase{};

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

```
// 局部特化
template <int tag>
struct Switch <tag, NilCase>
{
    typedef NilCase Result;
};
```

控制结构

类似地，可以分别给出更符合直觉的、结构性更好的For、While、Do-While实现。

最早提出编译期控制思想并给出雏形实现的是Todd Veldhuizen，参见[注1]。Krysztof Czarnecki, Ulrich Eisenecker则实现了更一般化的编译期结构（如刚才展示的第二个版本的Switch）参见[注12]。

数据结构

可以使用嵌套模板实现复杂的编译期数据结构（编译期容器），其中可以容纳整数和类型。

考察两个例子：一个序列，为Loki库中的Typelist；一个是二叉树（或树的二叉树表示）结构。

Boost MPL库中定义有vector、deque、list、set以及map等序列，它们都是编译期数据结构。

数据结构

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

```
// 哨卫类型
class NullType {};
```

```
// 计算长度
// 主模板
template <typename TList>
struct Length;
// 局部特化
template <typename T, typename U>
struct Length<Typelist<T, U> >
{
    enum { value = 1 + Length<U>::value };
};
// 完全特化
template <>
struct Length<NullType>
{
    enum { value = 0 };
};
```

一个typelist的长度等于tail的长度加1

```
// 示例
typedef Typelist<char, Typelist<int, NullType> > T;
cout << Length<T>::value << endl; // 2
```

数据结构

```
// BTree
const int LEAFVALUE = -1; // 叶子节点值

// 叶子节点
struct BTLeaf
{
    enum { Value = LEAFVALUE };
    typedef BTLeaf Left;
    typedef BTLeaf Right;
};

template <int N, typename Left_ = BTLeaf, typename Right_ = BTLeaf>
struct BTree // BTNode
{
    enum { Value = N };
    typedef Left_ Left;
    typedef Right_ Right;
};
```

数据结构

```
// 判树是否为空

// 主模板
template <typename BTree>
struct IsEmpty;

// 局部特化
template <int N, typename Left_, typename Right_>
struct IsEmpty<BTree<N, Left_, Right_> >
{
   enum { Result = false };
};

// 完全特化
template<>
struct IsEmpty<BTree<LEAFVALUE, BTLeaf, BTLeaf> >
{
   enum { Result = true };
};
```

数据结构

```
// 示例

typedef BTree<LEAFVALUE> tree1;
typedef BTree<1> tree2;
typedef BTree<1, BTree<2>, BTree<3> > tree3;

int main()
{
    cout << IsEmpty<tree1>::Result << endl; // 1
    cout << IsEmpty<tree2>::Result << endl; // 0
    cout << IsEmpty<tree3>::Result << endl; // 0
}
```

等价于：typedef BTree<1, BTLeaf, BTLeaf> aTree2;

数值计算

由于模板元编程最先是因为数值计算而被发现的，因此早期的研究工作主要集中于数值计算方面，先锋是Todd Veldhuizen和Blitz++库。元编程在该领域最早的应用是实现“循环开解（Unroll Loop）”。其他库（例如MTL、POOMA等）也采用了这种技术。

数值计算领域还有很多重要的模板元编程（相关）技术，例如“表达式模板（Expression Templates）”（见[注2]）、“部分求值（Partial Evaluation）”（见[注3]）等。

数值计算

一个经典的循环开解例子：计算向量点积

```
// 主模板
template <int DIM, typename T>
struct DotProduct
{
    static T Result(T* a, T* b)
    {
        return *a * *b  +  DotProduct<DIM-1, T>::Result(a+1, b+1);
    }
};
```

算法思想：向量a和b的点积=向量a和b首元素的乘积+剩余维度向量之点积

```
// 局部特化
template <typename T>
struct DotProduct<1, T>
{
    static T Result(T* a, T* b)
    {
        return *a * *b;
    }
};
```

一维向量的情形

数值计算

```
// 示例
int main()
{
    int a[5] = { 1, 2, 3, 4, 5 };
    int b[5] = { 6, 7, 8, 9, 10};
    cout << DotProduct<5, int>::Result(a, b) << endl; // 130
}
```

值得一提的一个模板元编程陷阱.....

```
// 循环开解过程
DotProduct<5,int>::result(a,b)
= *a * *b + DotProduct<4,int>::result(a+1,b+1)
= *a * *b + *(a+1) * *(b+1) + DotProduct<3,int>::result(a+2,b+2)
= *a * *b + *(a+1) * *(b+1) + *(a+2) * *(b+2) +
DotProduct<2,int>::result(a+3,b+3)
= *a * *b + *(a+1) * *(b+1) + *(a+2) * *(b+2) + *(a+3) * *(b+3) +
DotProduct<1,int>::result(a+4,b+4)
= *a * *b + *(a+1) * *(b+1) + *(a+2) * *(b+2) + *(a+3) * *(b+3) +
*(a+4) * *(b+4)
```

数值计算

长期以来，科学计算领域一直是Fortran的天下，采用运行期C++实现的算法太慢而无法适应数值计算的要求，利用元编程、表达式模板以及更好的编译器、优化器，现代C++也可以很好地满足数值计算的要求。

Todd Veldhuizen对C++和Fortran在科学计算领域的性能表现作了比较（参见[注4]）。

类型计算

实践证明，对于现代C++编程而言，元编程最大的用场本并不在于编译期数值计算，而是用于类型计算（type computation）（及相关领域）。通过类型参数、模板参数、typedef、枚举（或静态整型常量）以及内嵌类（模板）成员等，借助于灵活的类模板特化能力，模板元编程在类型计算方面可以释放出极大的能量。

类型计算

```
// 仅声明
struct Nil;

// 主模板
template <typename T>
struct IsPointer
{
    enum { Result = false };
    typedef Nil ValueType;
};

// 局部特化
template <typename T>
struct IsPointer<T*>
{
    enum { Result = true };
    typedef T ValueType;
};
```

```
// 示例
int main()
{
    cout << IsPointer<int*>::Result << endl;
    cout << IsPointer<int>::Result << endl;
    IsPointer<int*>::ValueType i = 1;
    //IsPointer<int>::ValueType j = 1;
    // 错误：使用未定义的类型Nil
}
```

可以依样实现出IsReference、IsClass、IsFloat、IsMemberPointer等元函数，事实上，Boost Type Traits库提供了数十个类似的元函数以及像add_reference这样的低级类型操纵元函数，用于侦测、处理类型的基本属性。该库已经被纳入C++标准委员会技术报告（Technical Report），这预示着它将会进入C++0x标准。

代码生成

前述的"循环开解"实际上就是一种代码生成机制，但模板元编程代码生成机制的作用并不局限于数值计算领域。

```
struct A
{
    static void execute()
    {
        cout << "A" << endl;
    }
};

struct B
{
    static void execute()
    {
        cout << "B" << endl;
    }
};
```

等价

```
int main()
{
    IfThenElse<(true), A, B>::ResultType::execute();
}
```

模板元程序充任"高级的预处理指令"

代码生成

前述的"循环开解"实际上就是一种代码生成机制，但模板元编程代码生成机制的作用并不局限于数值计算领域。

此例摘自Loki&MCD

```
struct Nil; // 仅声明

struct Empty{};
```

```
// Typelist
template <class H, class T>
struct TypeList
{
    typedef H Head;
    typedef T Tail;
};
```

```
template<class, class>
struct ScatterHierarchyTag;

template <class TList, template <class> class Unit>
struct GenScatterHierarchy;
//见下页.....
```

模板也是一种元数据

代码生成

```
// ……接上
template <class T1, class T2, template <class> class Unit>
struct GenScatterHierarchy<TypeList<T1, T2>, Unit>
    : public GenScatterHierarchy<ScatterHierarchyTag<T1, T2>, Unit>
    , public GenScatterHierarchy<T2, Unit>
{};

template <class T1, class T2, template <class> class Unit>
struct GenScatterHierarchy<ScatterHierarchyTag<T1, T2>, Unit>
    : public GenScatterHierarchy<T1, Unit>
{};

template <class AtomicType, template <class> class Unit>
struct GenScatterHierarchy : public Unit<AtomicType>
{};

template <template <class> class Unit>
struct GenScatterHierarchy<Nil, Unit>
{};
```

将一个非Typelist的原子类型传给Unit

处理Nil类型的情况——什么都不做

代码生成

```
// 自定义类型Test
struct Test
{
    enum {Value = 100};
};
template <typename T>
struct Holder
{
    T Value;
    static void f()
    {
        cout << sizeof(T) << endl;
    }
    // ...
};
// 定义一个包含有char、int、Test和float的Typelist
typedef TypeList<char, TypeList<int, TypeList<Test, TypeList<float, Nil> > > > CIRF;

typedef GenScatterHierarchy<CIRF, Holder> SH;
```

这个Holder模板决定了生成的各个类的能力

代码生成

```
int main()
{
    SH sh;
    cout << (dynamic_cast<Holder<char>&>(sh).Value = 'a') << endl;
    cout << (dynamic_cast<Holder<int>&>(sh).Value = 1) << endl;
    cout << (dynamic_cast<Holder<float>&>(sh).Value = 3.14f) << endl;
    cout << (dynamic_cast<Holder<Test>&>(sh).Value.Value) << endl;

    // cout << (dynamic_cast<Holder<double>&>(sh).Value = 3.14) << endl;
}
```

GenScatterHierarchy将给定的TypeList中的每一个类型施加于一个由用户提供的基本模板Holder上，从而产生一个类层次结构。换句话说，生成的各类的能力，取决于客户提供的Holder模板的能力。

示例中生成了一个多重继承类层次结构，Holder<char>, Holder<int>, Holder<float>, Holder<Test>之间没有任何关系，但它们都是SH的基类，即所产生的SH层次结构其实相当于：

class SH: public Holder<char>, public Holder<int>, public Holder<Test>, public Holder<float>;

代码生成

利用元编程生成类层次结构最大的好处在于其灵活性：TypeList是可扩展的，其长度不但可以任意定制，而且对其更改后SH可以自适应地产生新的代码。Loki库中的Abstract Factory泛型模式即借助于这种机制实现在不损失类型安全性的前提下降低对类型的静态依赖性。

进一步了解typelist和相关的代码生成技术，以及Abstract Factory泛型模式，参见[注11]和[注15]。

断言和契约

可以利用元编程技术实现编译期断言和编译期约束。

断言用于指定程序中某些特定点的条件应为“真”。如果该条件不为真，则断言失败，程序执行中断。大量使用断言可以在开发期捕捉许多错误。当然，若有可能，在编译期抓住错误更好。触发编译期断言的结果是导致程序无法通过编译。

编译期断言的实现方式并非仅限于模板元编程一种。

断言和契约

```
//主模板
template<bool>
struct StaticAssert;

// 完全特化
template<>
struct StaticAssert<true>
{};

// 辅助宏
#define STATIC_ASSERT(exp)\
{ StaticAssert<((exp) != 0)> StaticAssertFailed
}

int main()
{
   STATIC_ASSERT(0>1);
}
```

声明STATIC_ASSERT宏是为了方便使用，否则用户如果写StaticAssert<*false*>;在有些编译器（例如GCC和Borlad C++）中编译报错，在另外一些编译器（例如VC++和Digital Mars）中则可以编译通过。也就是说，单单"提及"一下StaticAssert是不保险的，要强迫它完全实例化才能保证触发断言，例如StaticAssert<false> S;这看上去有些臃怪，因此我们把它封装到宏里：StaticAssert<((exp) != 0)> StaticAssertFailed;

断言和契约

命名为StaticAssertFailed是为了便于在生成的编译错误信息中看出触发了一个静态断言，例如在GCC中生成如下错误信息：

```
sa.cpp:13: error: aggregate `StaticAssert< false> StaticAssertFailed'
has incomplete type and cannot be defined
```

这个错误信息还有改善的余地，你可以考察Boost和Loki中的静态断言库/组件，它们的实现更到位。

一个新的关键字static_assert极有可能被加入C++0x，其作用正如StaticAssert模板。

注意：表达式必须能够在编译期进行求值，无法对运行期表达式进行求值。

断言和契约

契约式设计（Design by Contract）要求为组件指定“契约”，这些契约会在程序运行过程中的某些特定点被强制执行。编译期契约也被称为约束（constraints）。C++虽然不直接支持约束，但是我们可以手工实现这项技术。

例如，结合运用上述StaticAssert和前面编写的IsPointer，可以实现一个约束：

```
#define CONSTRAINT_MUST_BE_POINTER(T) \
  STATIC_ASSERT(IsPointer<T>::Result != 0)
```

断言和契约

在以下类模板中，通过将该约束放在析构函数中，通常可以保证模板参数T必须是一个指针类型：

```
template <typename T>
class Test
{
public:
    ...
    ~Test()
    {
      // 只要该类的实例被创建，约束就会发挥作用
      CONSTRAINT_MUST_BE_POINTER(T);
    }
};
```

我们没有将它放置于构造函数中，因为构造函数可能不止一个……

```
int main()
{
    Test<int*> r; // OK
    Test<int> d;  // 违反约束，编译报错
}
```

库

高质量的库可以为开发可移植、高性能的应用提供良好的基础，一个经过缜密设计和测试的库具有良好的复用性，可以屏蔽平台之间的差异，可以使程序员专注于自己的业务开发。

C++模板的语法较复杂，一些惯用法应该采用库的方式提供。将这些复杂性封装于库中，并暴露给用户友好的接口，是每一个模板库开发者的责任，因为再复杂的库都应该是"面向用户"的。库中出现大量的繁杂的代码是可以接受的，因为这样的工作只需做一次，而所有库的用户均可从中受益。

知名的模板元编程库有Loki [注14]、Boost （元编程库）[注13]、Blitz++ [注15]以及MTL [注16]等。

库

Blitz++ 核心库的开发者是模板元编程技术的创始人Todd Veldhuizen，可以说是最早利用模板将运行期计算转移至编译期的库，主要提供了对向量、矩阵等进行处理的线性代数计算。长期以来，科学计算一直是Fortran77/90的地盘，而Blitz++利用元编程（以及表达式模板等）技术可以获得和Fortran77/90媲美的效率（参见[注4]）。

Loki 将模板元编程在类型计算方面的威力应用于设计模式领域，利用元编程（以及其他一些重要的设计技术）实现了一些常见的设计模式之泛型版本。

库

Boost 元编程库目前主要包含MPL、Type Traits和Static Assert等库。 Static Assert和Type Traits用作MPL的基础。

MPL是一个通用的模板元编程框架，它仿照STL提供了编译期算法、序列、迭代器等元编程组件。它为普通程序员进行元编程提供了高级抽象，使得元编程变得容易、高效、富有乐趣。

Boost Type Traits库包含一系列traits类，用于萃取C++类型特征。另外还包含了一些转换traits（例如移除一个类型的const修饰符等）。

Boost Static Assert库用于编译期断言，如果评估的表达式编译时计算结果为true，则代码可以通过编译，否则编译报错。

DSEL

对于领域特定的嵌入式语言（domain-specific embedded language, DSEL）的设计者而言，模板元编程技术是一件利器。这里的DSEL就是库，例如一些图形或矩阵计算库都可被认为是一种小型语言：其接口定义语法，其实现定义语义。它们均提供了领域特定的符号、构造和抽象。

利用模板元编程技术构建的DSEL，高效且语法接近于从头构建的语言。由于这种DSEL采用纯粹的C++编写，与使用独立的DSL相比，无需再使用专门的编译器、编辑器等工具，从而可以消除跨语言边界所需付出的代价。

如欲进一步了解采用C++模板元编程实现DSEL，参见[注9]。

DSEL

摘自Boost Lambda库的几个例子：

```
int a[5][10];
int i;
for_each(a,
        a+5,
        for_loop(var(i)=0, var(i)<10, ++var(i), _1[var(i)] += 1)
        );
```

```
std::for_each(v.begin(), v.end(),
            (switch_statement(_1,
            case_statement<0>(std::cout << constant("zero")),
            case_statement<1>(std::cout << constant("one")),
            default_statement(cout << constant("other: ") << _1)
            ),
            cout << constant("\n")
            )
            );
```

DSEL

```
for_each(
        a.begin(), a.end(),
        try_catch(
        bind(foo, _1),                  // foo may throw
        catch_exception<foo_exception>
        (
           cout << constant("Caught foo_exception: ")
              << "foo was called with argument = " << _1
        ),

        catch_exception<std::exception>
        (
           cout << constant("Caught std::exception: ")
              << bind(&std::exception::what, _e),
           throw_exception(bind(constructor<bar_exception>(), _1)))
        ),
        catch_all((cout << constant("Unknown"), rethrow())
        )
        )
        );
```

结语

模板元程序与常规代码结合使用时，此时源代码包含两种程序：常规C++运行期程序和编译期运行的模板元程序。当被编译器解释时，模板元程序可以生成高效的代码，从而可以大幅提高最终应用程序的运行效率。通过将计算从运行期转移至编译期，在结果程序启动之前做尽可能多的工作，最终获得速度更快的程序。

模板元编程也有一些局限性，使用模板元编程时（尤其使用模板元编程进行数值计算时）存在一些注意事项……

代码的可读性较差。

调试困难：元程序执行于编译期，没有用于单步跟踪元程序执行的调试器（用于设置断点、察看数据等）。程序员可做的只能是等待编译过程失败，然后人工破译编译器倾泻到屏幕上的错误信息。

结语

编译时间延长：元程序被编译器解释执行的，C++编译器并不是一个好的解释器。

结果程序性能未必一定最优化："循环开解"技术要有选择地使用，具体获得的效果必须进行评测。倘若代码展开导致程序尺寸过大，可能会降低cache的命中率，未必会带来性能上的提高。

编译器局限性：模板的实例化通常要占用不少编译器资源，大量的递归实例化会迅速拖慢编译器甚至耗尽可用资源。

可移植性较差：对于模板元编程使用的高级模板特性，不同的编译器的支持度不同。

资源

以下是一些C++模板元编程资源，它们或者为本幻灯片的制作提供了参考素材，或者提供了延伸知识。

文章

[1] Todd Veldhuizen, *Template Metaprograms* , http://osl.iu.edu/~tveldhui/papers/Template-Metaprograms/meta-art.html

[2] Todd Veldhuizen, *Expression Templates*, http://osl.iu.edu/~tveldhui/papers/Expression-Templates/exprtmpl.html

资源

[3] Todd Veldhuizen, *C++ templates as partial evaluation*, http://osl.iu.edu/~tveldhui/papers/pepm99/.

[4] Todd Veldhuizen, *Scientific Computing: C++ versus Fortran* , http://osl.iu.edu/~tveldhui/papers/DrDobbs2/drdobbs2.html

[5] Erwin Unruh, Prime numbers(Primzahlen - Original), http://www.erwin-unruh.de/primorig.html .

[6] Erwin Unruh, Prime numbers(Primzahlen), http://www.erwin-unruh.de/Prim.html .

[7] Edward Rosten, *Floating point arithmetic in C++ templates* http://mi.eng.cam.ac.uk/~er258/code/fp_template.html.

资源

书籍

[8] David Vandervoode and Nicolai M. Josuttis. *C++ Templates: The Complete Guide.* Addison-Wesley

[9] David Abrahams, Aleksey Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*, Addison-Wesley Professional.

[10] Matthew Wilson, *Imperfect C++ : Practical Solutions for Real-Life Programming*, Addison-Wesley Professional.

[11] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley

[12] Krysztof Czarnecki, Ulrich Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley Professional

资源

库

[13] Boost http://www.boost.org/

[14] Blitz++ http://www.oonumerics.org/blitz/

[15] Loki http://sourceforge.net/projects/loki-lib/

[16] MTL http://www.osl.iu.edu/research/mtl/

[17] FC++ http://www.cc.gatech.edu/~yannis/fc++/

The End