Catálogo de Técnicas de Meta-Programação por Templates em C++

Sérgio Vale e Pace

Departamento de Ciência da Computação - PUC Minas

svpace@gmail.com

Orientador: Marco Túlio de Oliveira Valente

Departamento de Ciência da Computação - PUC Minas

mtov@pucminas.br

Abstract. *C++ Template Meta-programming is the set of programming techniques that uses the template instantiation mechanism as interpreter to evaluate meta-programs at compile-time. This work aims to document some basic template meta-programming techniques providing basic/intermediary level reference material for programmer who wish to apply or develop such techniques.*

Resumo. *Meta-Programação por Templates em C++ é o conjunto de técnicas de programação que utilizam o mecanismo de instanciação de templates como interpretador para a avaliação de meta-programas em tempo de compilação. Esse trabalho objetiva documentar algumas técnicas básicas de meta-programação por templates fornecendo material de referência de nível básico/intermediário para programadores que desejam aplicar ou desenvolver essas técnicas.*

1. Introdução

Meta-programação é a ação de desenvolver meta-programas, que geram, modificam ou instrumentam um programa resultante. Meta-programas são descritos através de uma meta-linguagem que permite a manipulação dos elementos simbólicos da linguagem base, na qual o programa resultante será descrito. Como ilustra a figura 1, na programação convencional, temos um programa que irá dirigir um processamento que produzirá uma saída. Utilizando meta-programação temos um meta-programa que irá dirigir o processamento que produzirá o programa.

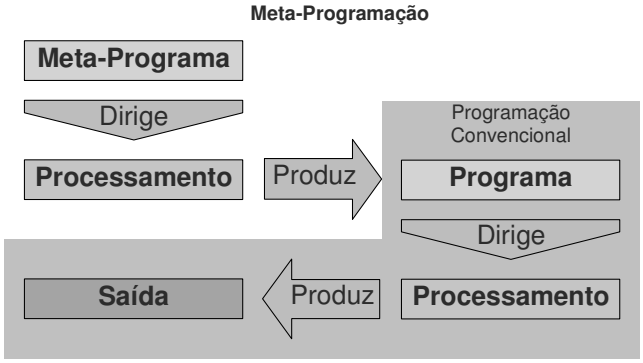

Figura 1 - Meta-programação e Programação Convencional

O processo de compilação usado ao se traduzir programas escritos em uma linguagem de alto nível para programas em linguagem de máquina é uma forma de meta-programação e os benefícios observados ao se desenvolver programas utilizando linguagens de alto nível ao invés de utilizar linguagem de máquina são fundamentalmente os mesmos observados na utilização de técnicas de meta-programação em geral. Meta-programas são executados em tempo e contexto diferentes de seus programas resultantes [Cz01], oferecendo assim, principalmente:

- Maior nível de abstração e estruturação lógica;
- Maior escalabilidade e extensibilidade;
- Redução do volume de código necessário;
- Redução do esforço de manutenção;
- Possibilidades de otimização;
- Possibilidade de pré-executar tarefas.

Meta-programação por *Templates* (*Template Meta-programming*, ou TMP) é o conjunto de técnicas de programação que utilizam o mecanismo de instanciação de *templates* com o objetivo de prover capacidade de meta-programação a uma linguagem de programação.

A primeira linguagem de programação onde tais técnicas foram aplicadas, devido a ser uma das primeiras linguagens a suportar os recursos necessários para isto, foi a linguagem C++ [IS98]. Atualmente a linguagem C++ ainda é a principal linguagem de desenvolvimento de TMP, porém existem esforços, com diferentes objetivos e graus de sucesso, para adaptar essas técnicas a outras linguagens de programação, como Haskell [Sh02] e Java [Ve00].

A primeira demonstração de uma técnica de TMP de que se tem registro foi realizada por Erwin Unruh [Un94], e apresentada a vários membros do comitê ANSI/ISO de padronização da linguagem C++ [Ve95]. O programa calcula todos os números primos menores que uma dada constante. Ironicamente, o código não é um programa C++ válido e produz sua saída através das mensagens de erro do compilador.

Embora esse programa não seja de grande interesse prático, ele demonstrou que o mecanismo de *templates* da linguagem C++ poderia ser usado para executar computações em tempo de compilação.

Posteriormente o artigo de Todd Veldhuizen [Ve95] lançou as bases para aplicações práticas utilizando TMP, e, mais tarde, o mesmo autor demonstrou que o mecanismo de instanciação de *templates* da linguagem C++ forma uma linguagem de programação Turing completa [Ve03], ou seja, a meta-linguagem usada para TMP é uma linguagem de poder computacional equivalente a qualquer outra linguagem de programação de uso geral.

A partir daí um número crescente de técnicas de TMP tem sido desenvolvidas, sendo inclusive algumas delas consideradas pelo comitê de padronização ISO/IEC para integrar as próximas versões da biblioteca padrão de C++ [IS03].

O objetivo desse trabalho é catalogar as técnicas fundamentais de TMP, reduzindo-as a seu mínimo funcional, ou seja, reduzindo ou eliminando elementos auxiliares normalmente empregados para aumentar a robustez e a portabilidade dos códigos usados em bibliotecas de TMP, como [BMPL] e [LOKI], e organizá-las de maneira coerente, de modo a criar uma forma reduzida de manual de referência da meta-linguagem de programação por *templates*. O capítulo 2 descreve os conceitos básicos e peculiaridades da TMP em C++ e analisa as características da linguagem que possibilitam o desenvolvimento de meta-programas. O capítulo 3 descreve as varias técnicas de TMP. O capítulo 4 faz um estudo de caso de uma aplicação de TMP. O capítulo 5 apresenta a conclusão e propõe trabalhos futuros.

2. Meta-Programas por *Template*

Um exemplo clássico de TMP, apresentado inicialmente em [Ve95], é o cálculo do fatorial de um número em tempo de compilação:

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

O ponto fundamental desse exemplo está na definição recursiva dada a **`Fact<N>::value`** que obriga o compilador a avaliar **`Fact<N-1>::value`** e assim sucessivamente até que a especialização do *template* para **`Fact<1>`** termine a recursão, ou seja, internamente o compilador irá avaliar a definição recursiva seguindo os passos:

```
int Example = Fact<3>::value;
int Example = 3 * Fact<2>::value;
int Example = 3 * 2 * Fact<1>::value;
int Example = 3 * 2 * 1; // Otimização de expressão constante
int Example = 6;
```

Essa técnica combinada com programação C++ convencional permite implementação de algoritmos altamente especializados, reduzindo ao máximo o volume de trabalho a ser realizado em tempo de execução.

2.1. Peculiaridades da Meta-Programação por *Templates*

A TMP, assim como qualquer técnica de programação avançada, possui peculiaridades que devem ser observadas para um uso eficiente da técnica. Algumas das peculiaridades mais significativas da TMP são explicadas a seguir.

2.1.1. Elementos estáticos

Um meta-programa é avaliado em tempo de compilação. Essa é a característica mais significativa da TMP e é a principal responsável por seus benefícios e limitações. Como resultado disso, todo meta-programa deve estar completamente definido no momento em que o programa é compilado. Por isso um meta-programa é composto basicamente de elementos estáticos da linguagem como definições de tipos (**`class`**, **`struct`**, **`union`**, **`typedef`**, **`enum`**), funções livres, funções e variáveis membro estáticas. Um meta-programa não é capaz de manipular diretamente os elementos dinâmicos da linguagem como objetos, variáveis ou entrada e saída, pois esses elementos só estarão disponíveis durante a execução do sistema.

2.1.2. Programação Funcional

No paradigma de programação imperativa na qual a linguagem C++ se baseia, um programa é concebido como uma seqüência de instruções que modificam de alguma forma o estado do sistema. Já no paradigma da programação funcional usado em linguagens como Haskell e Lisp, um programa é concebido como um grupo de declarações, que relacionam entradas às saídas específicas.

Apesar da linguagem C++ ser uma linguagem imperativa, esse paradigma não é adequado à TMP, pois, como foi dito anteriormente, todo meta-programa por templates é estático por definição e, portanto o conceito de mudança de estado no qual a programação imperativa se baseia não é compatível com a natureza estática da TMP.

Em [Th99] é feita uma análise comparativa entre o paradigma funcional e imperativo. A diferença no conceito de variável nos dois paradigmas é particularmente interessante para a TMP. No paradigma imperativo, uma variável é um elemento que pode assumir diversos valores durante a sua vida útil. Já no paradigma funcional uma variável é um elemento cujo valor não é conhecido, portanto "variáveis em programas funcionais não variam" [Th99].

Assim, a meta-linguagem usada na TMP segue o paradigma funcional, e as técnicas e algoritmos usados são baseados principalmente em técnicas e algoritmos usados em linguagens funcionais. Essa característica é um dos principais motivos pelo qual TMP é considerada uma técnica complexa por muitos programadores C++.

2.1.3. *Code Bloat*

Uma característica marcante do mecanismo de *templates* em C++ é o fato que cada instância de um *template* gera um tipo de dados distinto, com implementação distinta de

outras instâncias do mesmo *template*. Conseqüentemente, ao se fornecer N conjuntos de parâmetros distintos a um *template* serão geradas N implementações distintas desse *template*. Esse é um problema conhecido na comunidade da programação genérica como *Code Bloat* (Código Inchado).

Uma característica desejável de um meta-programa é que ele seja "consumido" durante o processo de compilação e o código executável gerado não contenha resíduos do meta-programa. Nem sempre é possível criar meta-programas que não gerem nenhum resíduo, mas um meta-programador deve sempre buscar esse objetivo. Deve ser dada especial atenção às técnicas que exijam recursões, tendo em vista que nesse tipo de técnica é utilizado um grande número de instâncias de *template*.

Para prevenir ou reduzir o *Code Bloat* é importante que as estruturas usadas em TMP sejam tão desprovidas de conteúdo dinâmico quanto possível, contendo somente elementos que não irão gerar código executável ou consumir espaço de armazenamento em tempo de execução, como tipos definidos (**`typedef`**), enumerações (**`enum`**) e funções *inline* (ver seção 2.2.4).

2.1.4. Dependência do Compilador

Apesar de a primeira edição do padrão ISO para a linguagem C++ ter sido publicada há mais de 10 anos [IS98], o grau médio de aderência ao padrão apresentada por compiladores C++ é baixo. Técnicas de TMP tendem a levar o compilador ao limite e podem não ser suportadas por compiladores com baixo grau de aderência ao padrão.

Devido a isso, as bibliotecas de TMP, com freqüência, têm que empregar diversos artifícios para contornar as deficiências dos compiladores, o que tende a tornar seu código denso e difícil de ler.

Para esse trabalho, será utilizado como compilador padrão o GCC C++ [GNUC], GNU Compiler Collection C++, que é o compilador C++ do projeto GNU. Esse compilador foi selecionado por ser um projeto aberto e livre, por estar disponível em diversas plataformas, e por ser considerado pela comunidade C++ como um dos compiladores mais aderentes ao padrão ISO C++ [IS98].

2.2. Recursos da Linguagem

Com o objetivo de generalizar a discussão sobre as técnicas de TMP para outras linguagens além da C++, são enumerados a seguir recursos que estão disponíveis em C++ e que são requisitos para a meta-linguagem de TMP. Teoricamente, as técnicas apresentadas poderiam ser implementadas em qualquer linguagem de programação que suporte esses recursos.

2.2.1. *Templates*

A linguagem deve permitir parametrizar um tipo de dados baseado em outros tipos e valores[1]. Um *template* é fundamentalmente um meta-tipo, que gera uma família de tipos de dados semelhantes que são instanciados em tempo de compilação.

2.2.2. Referência para Parâmetro de *Template*

A linguagem deve permitir criar uma referência aos parâmetros de *template* de modo que estes possam ser acessados posteriormente.

Em C++, referências para parâmetros de valor podem ser construídas tanto com enumerações quanto com variáveis membro estáticas, porém o uso de variáveis estáticas não é recomendado, pois, ao contrário das enumerações, esta exigirá espaço de armazenamento em memória em tempo de execução [Va02].

2.2.3. Especialização de *Templates*

A linguagem deve permitir especificar implementações alternativas para um conjunto de parâmetros de *template* específicos. Embora ainda hoje alguns compiladores não suportem esse recurso foi demonstrado em [Cz00] que é possível simular especialização de templates nesses compiladores.

2.2.4. Funções *Inline*

A linguagem deve permitir criar funções que, depois de compiladas, serão inseridas nos locais que as invocam. Desse modo, ao ser invocada, uma função *inline* não causa um desvio no fluxo do programa como ocorre com funções convencionais. Conseqüentemente, funções *inline* não têm custo de invocação e não geram código nas classes que as definem.

Deve ser observado, porém que, de acordo com o padrão C++ [IS98], a palavra chave **`inline`** é apenas uma sugestão ao compilador e pode ser ignorada. Muitos compiladores possuem diretivas internas que permitem ao programador especificar que uma função será garantidamente *inline*. O compilador utilizado neste artigo irá gerar funções *inline* se utilizado o nível de otimização 3 (-O3) [GNUC].

3. Catálogo de Técnicas

O Formato do catálogo a seguir foi inspirado em "Padrões de Projeto" [Ga00], e adaptado às necessidades das técnicas de TMP. Cada técnica possui as seguintes seções:

1. **Nome:** Nome da técnica, uma descrição de seu objetivo e um conceito equivalente da programação convencional.
2. **Descrição:** Detalhamento da técnica e sua implementação
3. **Exemplo:** Um exemplo da utilização da técnica

[1] O termo "tipo de dados" será usado no decorrer desse trabalho em oposição ao termo "classe" comumente usado no paradigma orientado a objetos, por ser mais abrangente e para evitar a equivalência com a construção do tipo **`class`** do C++ uma vez que o termo deve designar construções do tipo **`class`**, **`struct`**, **`union`**, tipos primitivos, etc.

As técnicas catalogadas foram escolhidas e ordenadas baseadas em sua simplicidade, generalidade e em suas inter-relações, sendo que as primeiras técnicas são mais simples e de uso geral, servindo de alicerce para as demais.

Os nomes usados para as técnicas, campos, métodos e outros foram escolhidos baseados nos nomes mais comuns encontrados na literatura da área e de modo a manter coerência entre as técnicas, porém existem muitas variações, por vezes até em trabalhos distintos de um mesmo autor. Os nomes foram mantidos em inglês sempre que não houve uma tradução óbvia.

3.1. Bases

Criar uma entidade identificável associada a uma informação. O conceito equivalente em programação convencional é o de **variáveis**.

3.1.1. Descrição

Uma Base é um *template* que armazena referências para seus parâmetros, criando uma entidade identificável dentro do meta-programa, capaz de armazenar informações para uso futuro. Dentro do paradigma funcional, uma base é o correspondente a uma variável, ou seja, um nome identificável que possui dados associados.

O tipo de uma Base é análogo ao tipo de uma variável e deve ser adequado a natureza do dado que se deseja representar. Serão apresentadas a seguir algumas das Bases elementares mais usuais em TMP.

Bases Vazias. As Bases mais simples que se pode criar são as Bases vazias, também chamadas de marcadores, são Bases que não possuem qualquer dado associado. Como exemplo segue a implementação da chamada Base nula, que representa um elemento nulo ou inválido, definida da seguinte forma:

```
struct Nil {};
```

Bases vazias são apenas estruturas vazias, sua única característica relevante é seu nome que a identifica de maneira unívoca.

Bases de Tipo. Possuem um único parâmetro de *template* para um tipo de dados, e definem um membro chamado **`type`** que permite a recuperação posterior desse parâmetro:

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

Bases de Valor. Podem ser usadas para armazenar qualquer tipo de dados que possa ser convertido em um inteiro, como dados booleanos, enumerações e, obviamente, inteiros. Recebem como parâmetro o tipo e o valor do dado e permitem a recuperação deste através do membro **`value`**:

```
template<typename T, T V> struct Value { enum { value = V }; };
```

Bases Inteiras. É um caso específico das bases de valor, cujo tipo do dado é pré-definido como inteiro. Observe que outras Bases de Valor específicas podem ser criadas dessa mesma forma:

```
template<int V> struct Int : Value<int,V> {};
```

3.1.2. Exemplo

Considere uma função **`foo()`** que recebe um parâmetro enumerado do tipo **`Tasks`** e que baseado nele executa uma dentre varias tarefas. Utilizando programação convencional, esse código poderia ser escrito da seguinte forma:

```
enum Tasks { A, B, C };

Void foo(Tasks t) {
    switch(t) {
        case A: /* Tarefa A */; break;
        case B: /* Tarefa B */; break;
        case C: /* Tarefa C */; break;
};

foo(A);
```

Se considerarmos que a seleção da tarefa é sempre feita de forma estática, é possível utilizar uma Base de Valor enumerada para otimizar o código anterior, substituindo a construção do tipo *switch-case* avaliada em tempo de execução pela sobrecarga de função que será avaliada em tempo de compilação:

```
Void foo(Value<Tasks, A>) { /* Tarefa A */ };
Void foo(Value<Tasks, B>) { /* Tarefa B */ };
Void foo(Value<Tasks, C>) { /* Tarefa C */ };

foo(Value<Tasks, A>());
```

3.2. Meta-funções

Obter, em tempo de compilação, o resultado de uma computação aplicada a um dado conjunto de parâmetros. O conceito equivalente em programação convencional é o de **funções**.

3.2.1. Descrição

As Meta-funções, juntamente com as Bases, são os elementos construtivos básicos da TMP. Meta-funções são *templates* contendo expressões e construções que, durante o processo de instanciação, são avaliadas e consumidas, realizando nesse processo computações sobre seus parâmetros, resultando na formação de uma Base que contém o resultado dessas computações.

Em sua forma mais simples, Meta-funções simplesmente utilizam operações aritméticas ou de tipo, armazenando os resultados em Bases adequadas. Como exemplo observe a meta-função **`Times2<V>`**, que dobra o valor de seu parâmetro e o armazena em uma base de inteira:

```
template<int V> struct Times2 : Int<2*V> {};
```

Meta-Funções podem ser descritas através de herança, ou através de redefinição. Por herança, as meta-funções herdam os elemento de outras meta-funções ou bases, como pode ser visto no exemplo acima. Por redefinição, todos os elementos de interesse devem ser redeclarados dentro da meta-função. O exemplo anterior pode ser reescrito, utilizando redefinição, da seguinte forma:

```
template<int V> struct Times2 { enum { value = 2 * V }; };
```

A Meta-Funções como a **`Times2`** não são particularmente interessantes, pois oferecem pouca ou nenhuma vantagem em relação à utilização direta da expressão. Porém utilizando especialização de *templates*, Meta-funções passam a ser capazes de realizar desvios condicionais, operações recursivas e outras operações mais complexas.

3.2.2. Exemplo

A linguagem C++ inclui facilidades para a representação direta de números nas bases octal, decimal e hexadecimal. Porém ao se trabalhar com *bits* isolados, freqüentemente é necessário representar números em base binária e a linguagem C++ padrão não possui suporte para a representação direta de números binários. Através de uma meta-função é possível implementar esse recurso:

```
template<unsigned int N> struct Bin :
    Int<( N % 10 ) + ( 2 * Bin<( N / 10 )>::value)> {};
template<> struct Bin<0> : Int<0> {};
```

A meta-função **`Bin<>`** reinterpreta em tempo de compilação um número em base decimal fornecido no parâmetro do *template* como se fosse um número binário, retornando o resultado em uma base inteira:

```
int A = Bin<1010>::value;  // A = 10
int B = Bin<1000>::value;  // B = 8
int C = -Bin<10>::value;   // C = -2
```

3.3. *Wrappers*

Criar bases e meta-funções que encapsulam dados nativos e suas operações. O conceito equivalente em programação convencional é o de **operações nativas**.

3.3.1. Descrição

Um recurso bastante útil das bases de valor e suas derivadas é a possibilidade de utilizar os operadores nativos da linguagem para realizar operações aritméticas, lógicas e comparações.

Para isso é necessário primeiro obter os elementos relevantes das bases envolvidas, depois executar as operações nativas apropriadas e em seguida gerar uma nova base para conter o resultado. Tudo isso tende a poluir o código e a torná-lo desnecessariamente mais complexo. A situação se agrava ainda mais quando é necessário misturar operações nativas e Meta-funções.

Para remediar essa situação, podemos encapsular operadores nativos em Meta-funções envoltórias (*Wrappers*) e assim simplificar e uniformizar a sintaxe utilizada nos meta-programas.

Como exemplo, considere a criação de *Wrappers* que simplifiquem a utilização de operações lógicas diretamente em bases de valor booleanas. É útil criar definições explícitas para os dois estados possíveis de um dado booleano, de maneira a simplificar seu uso. Para isso, definimos dois tipos derivados de **`Value`**:

```
typedef Value<bool,true> True;
typedef Value<bool,false> False;
```

Dados booleanos possuem três operações elementares a partir das quais outras podem ser derivadas: **`Not`**, **`And`** e **`Or`**. Essas Meta-funções podem ser implementadas utilizando *Wrappers* que encapsulam as operações nativas da linguagem:

```
template<typename B> struct Not : Value<bool,!B::value> {};
template<typename LHS, typename RHS> struct And :
    Value<bool, LHS::value && RHS::value> {};
template<typename LHS, typename RHS> struct Or :
    Value<bool, LHS::value || RHS::value> {};
```

A partir dessas três operações elementares a criação de operações derivadas é bastante direta. Como exemplo, a Meta-função **`Xor`** é implementada a seguir:

```
template<typename LHS, typename RHS> struct Xor :
    Or< And< Not<LHS>, RHS >, And< LHS, Not<RHS> > > {};
```

A mesma técnica pode ser usada para encapsular quaisquer operações nativas da linguagem, como por exemplo, Meta-funções aritméticas para bases de inteiras. Além disso a criação de *Wrappers* nos permite utilizar as operações aritméticas existentes na linguagem como funções de alta ordem no meta-programa.

3.3.2. Exemplo

Os *Wrappers* servem fundamentalmente ao propósito de simplificar o desenvolvimento de meta-programas, eliminando parte da burocracia sintática necessária para implementar uma Meta-função. Considere, como exemplo, a implementação da função combinação para bases de inteiras:

```
template<typename C, typename K> struct Comb : Int<
    Fact<C>::value /
    ( Fact<K>::value * Fact< Value<C::value – K::value> >::value )
> {};
```

Utilizando *Wrappers*, a implementação dessa Meta-função se torna mais simples e condizente com a sintaxe de um programa funcional:

```
template<typename C, typename K> struct Comb :
    Div< Fact<C>, Mul< Fact<K>, Fact< Sub<C,K> > > > {};
```

3.4. *Type Traits*

Obter e manipular informações sobre um tipo. O conceito equivalente em programação convencional é o de identificação de tipos (*reflection*).

3.4.1. Descrição

Através da especialização de *templates* podemos definir implementações distintas para um *template* baseado em seus parâmetros, o que torna possível especializar Meta-funções para identificar tipos específicos. Uma implementação bastante abrangente dessa técnica pode ser encontrada em [BTTL]. O exemplo a seguir mostra como determinar se um tipo é uma Base Vazia:

```
template<typename T> struct IsNil : False {};
template<> struct IsNil<Nil> : True {};
```

Um *template* pode ser parcialmente especializado. Isso nos dá a flexibilidade de manipular tipos de maneira polimórfica estabelecendo relações entre tipos parcialmente desconhecidos. Por exemplo, podemos determinar se um tipo é idêntico a outro mesmo sem conhecer os tipos de dados:

```
template<typename T1, typename T2> struct IsEq : False {};
template<typename T> struct IsEq<T,T> : True {};
```

De maneira semelhante, é possível criar Meta-funções para identificar diversas características de uma variável C++:

```
template<typename T> struct IsConst : False, Type<T> {};
template<typename T> struct IsConst<const T> : True {};

template<typename T> struct IsPtr : False , Type<T> {};
template<typename T> struct IsPtr<T*> : True, Type<T> {};

template<typename T> struct IsArray :
    False, Type<T> { enum { size = 0}; };
template<typename T, unsigned int N> struct IsArray<T[N]> :
    True, Type<T> { enum { size = N }; };
```

3.4.2. Exemplo

Ao produzir códigos genéricos é comum a necessidade de obter um ponteiro para um tipo de dados desconhecido. Porém, se o tipo em questão já é um ponteiro, este não precisa ser modificado, caso contrário será obtido um ponteiro para um ponteiro e não um ponteiro para um dado. Para resolver esse problema, podemos utilizar a técnica de *Type Traits* para identificar o tipo de dados especificado e então modificá-lo:

```
template<typename B, typename T> struct AddPtrAux : Type<T*> {};
template<typename T> struct AddPtrAux<True,T> : Type<T> {};
template<typename T> struct AddPtr : AddPtrAux<IsPtr<T>, T> {};
```

```
AddPtr<int>::type A;  // int* A;
AddPtr<int*>::type B; // int* B;
```

Observe que dessa forma **`AddPtr<>::type`** será um ponteiro para **`T`** somente se **`T`** já não é um ponteiro, garantindo assim que a base de tipo resultante contenha sempre um ponteiro para um dado e nunca para outra ponteiro.

3.5. Meta *If*

Permitir a utilização de seleções do tipo *if-then-else* em meta-programas. O conceito equivalente em programação convencional é o de **tomada de decisão**.

3.5.1. Descrição

Construções de tomada de decisão do tipo *if-then-else* são as mais simples e habituais estruturas de controle em programação imperativa. Em TMP, o principal mecanismo de tomada de decisão é a especialização de *templates*. A técnica Meta *If* busca criar uma solução genérica para a tomada de decisão, evitando a necessidade de especializar todos as Meta-funções que exijam tomada de decisão.

A implementação da Meta-função *If* é feita usando especialização parcial sobre um parâmetro de *template* booleano que retorna um dentre dois parâmetros fornecidos através de uma base de tipo:

```
template<bool B, typename Then, typename Else> struct If :
    Type<Then> {};
template<typename Then, typename Else> struct If<false,Then,Else> :
    Type<Else> {};
```

Desse modo, Meta-funções que de outra forma exigiriam duas ou mais declarações, podem ser escritas utilizando uma única declaração, resultando em uma notação mais familiar a programadores imperativos:

```
// Sem meta-if
template<bool B> struct Foo : Type<CaseTrue> {};
template<> struct Foo<false> : Type<CaseFalse> {};
// Com meta-if
template <bool B> struct Foo : If<B, CaseTrue, CaseFalse> {};
```

3.5.2. Exemplo

Considere uma classe **`container`** parametrizada com a seguinte assinatura:

```
template<typename T> class Container;
```

Deseja-se implementar nessa classe uma operação de atribuição de modo que o conteúdo de uma instância da mesma possa ser copiado para outra. Essa tarefa pode ser executada usando dois algoritmos distintos, **`DeepCopy`** e **`ShallowCopy`**.

ShallowCopy é mais eficiente em termos de tempo e espaço de armazenamento, porém ele só pode ser aplicado caso os elementos de **Container** sejam ponteiros ou referências para elementos constantes. Caso contrário, **DeepCopy** deverá ser utilizado.

A decisão sobre qual algoritmo será usado pode ser feita em tempo de compilação, pois o tipo dos elementos de **Container** é informado durante a instanciação do *template*.

Cria-se na classe **Container** um método **copy()** que utiliza a técnica *Type Traits* para identificar as características do tipo de dados, ou seja, através das Meta-funções **IsPtr<>**, **IsRef<>** e **IsConst<>** buscamos por ponteiros ou referencias para elementos constantes.

De posse dessas informações uma operação lógica será criada e seu resultado passado para uma implementação da técnica Meta *If* que irá selecionar um dentre os dois algoritmos implementados de modo que o método **copy()** possa invocá-lo a partir do método estático **apply()**:

```
template<typename T> struct DeepCopy {
    static inline void apply(Container<T>& src, Container<T>& dst)
    {...}
};
template<typename T> struct ShallowCopy {
    static inline void apply(Container<T>& src, Container<T>& dst)
    {...}
};
template<typename T> class Container {
    ...
    void copy(Container<T>& that) {
        If< And< IsConst<T>, Or< IsPtr<T>, IsRef<T> > >::value,
             ShalowCopy,
             DeepCopy
        >::type::apply(that, *this);
    }
   ...
};
```

Dessa forma ao invocar o método **copy** em qualquer instancia do *template* **Container**, será invocado o algoritmo mais apropriado para cada situação e isso ocorrerá de maneira transparente ao usuário do *template*.

3.6. Lista de Tipos

Criar uma estrutura de dados do tipo lista encadeada para tipos. O conceito equivalente em programação convencional é o de **lista encadeada**.

3.6.1. Descrição

A Lista de Tipos é o correspondente em TMP à estrutura de dados lista encadeada usada na programação convencional. Assim como as listas encadeadas convencionais, as Listas de Tipos são compostas por células que contêm os dados a serem armazenados e um apontador para a próxima célula da lista. Chamaremos essas células de Bases de Lista. Bases de Listas nada mais são que Bases de Tipo acrescidas de um membro auxiliar que irá apontar a próxima célula da lista. Para marcar o final da lista será usado um tipo nulo **`Nil`**:

```
template<typename T, typename N=Nil> struct LI : Type<T> {
    typedef N next;
};
```

Dessa forma, a criação de listas de tipo se dá pelo encadeamento de diversas Bases de Lista, como no exemplo a seguir:

```
typedef LI<signed char, LI<short, LI<long > > > Integers;
```

Adicionalmente se faz útil criar Meta-funções para manipular essas listas de maneira eficiente. A título de exemplo é mostrado a seguir os códigos para algumas das Meta-funções mais usuais para manipulação de Lista de Tipo:

`size<L>` obtém o tamanho da lista **`L`**:

```
template<typename L> struct Size :
    Value< ( 1 + Size<typename L::next>::value ) > {};
template<> struct Size<Nil> : Value<0> {};
```

`at<L,N>` obtém a **`N`**-ésima célula da lista **`L`**:

```
template<typename L, unsigned int N> struct At :
    At<typename L::next, N-1> {};
template<typename L> struct At<typename L,0> : L {};
```

`add<L,T>` cria uma nova lista adicionando o tipo **`T`** ao final da lista **`L`**:

```
template<typename L, typename T> struct Add :
    li< typename L::type, Add<typename L::next, T> > {};
template<typename T> struct Add<Nil, T> : li<T, Nil> {};
```

`invert<L>` inverte a ordem da lista **`L`**:

```
template<typename L> struct Invert :
      add<Invert<typename L::next>, typename L::type> {};
```

Implementações completas de Lista de Tipo podem ser encontradas em [LOKI] e [BMPL].

3.6.2. Exemplo

Considere uma família de sistemas em que uma dada função será composta a partir de diversos módulos. Independente de quais módulos serão utilizados, deve ser contabilizado o tempo gasto em cada módulo. Uma implementação trivial desse sistema é dada a seguir:

```
Void foo() {
    TimeLog::begin();
        Module1::exec();
    TimeLog::end();

    TimeLog::begin();
        Module2::exec();
    TimeLog::end();

    //...

    TimeLog::begin();
        ModuleN::exec();
    TimeLog::end();
};
```

Essa implementação apresenta dois problemas, um deles é a grande quantidade de código repetido, o outro é que a definição de quais módulos serão utilizados se encontra "espalhada" por toda a função dificultando a customização do sistema.

Uma vez que os módulos utilizados em cada sistema individual da família de sistemas são conhecidos em tempo de compilação, pode-se utilizar uma Meta-função que irá percorrer uma Lista de Tipos contendo a especificação dos módulos e acioná-los na devida ordem:

```
template<typename T> struct Foo {
    static inline void exec() {
        TimeLog::begin();
        T::type::exec();
        TimeLog::end();
        Foo<typename T::next>::exec();
    };
};
template<> struct Foo<Nil> {
    static inline void exec() {};
};
typedef Foo< LI<Module1, LI<Module2, ... LI<ModuleN>...> >
> myFoo;

myFoo::exec();
```

Observe que a solução meta-programada quando compilada irá gerar exatamente o mesmo código apresentado para a solução trivial, porém através da utilização de uma Lista

de Tipos foi possível eliminar a repetição de código assim como concentrar toda a configuração de módulos em um único bloco de código.

4. Estudo de Caso – SIUnits

SIUnits é uma biblioteca para computação baseada em unidades desenvolvida, em parte, sob os auspícios do projeto Zoom [ZOOM]. Esse estudo de caso é baseado na descrição da biblioteca SIUnits apresentada em [Br01].

O seguinte código implementa em C++ a fórmula que calcula a grandeza física velocidade a partir das também grandezas físicas distância (**`dist`**) e tempo (**`time`**):

```
double getSpeed(double dist, double time) { return( dist / time ); }
```

Uma regra a ser observada ao se manipular grandezas físicas é que cada valor deve ser acompanhado por sua unidade de medida. O código acima não traz nenhuma informação sobre as unidades de medida envolvidas tornando erros, como a troca da ordem dos parâmetros ou a utilização de um peso ao invés de distância simples de cometer e difíceis de detectar. Em um software científico de larga escala em que há centenas de formulas, a identificação desse tipo de erro pode ser bastante desafiadora.

A inclusão da análise dimensional, utilizando técnicas de tempo de execução, não é atraente devido ao seu custo em termos de desempenho. Observe que os requisitos básicos de um sistema de análise dimensional são fundamentalmente idênticos aos requisitos do sistema de tipos presente em qualquer linguagem de tipagem forte. Dessa forma considere o seguinte pseudo-código que é um refinamento do código anterior usando a SIUnits:

```
speed<> getSpeed(distance<> dist, time<> time) { return( dist / time ); }
```

Nessa implementação o erro lógico é prontamente reportado pelo compilador, pois o resultado do operador multiplicação aplicada a um operando do tipo **`distance<>`** e um operando do tipo **`time<>`** não resulta em uma instância de **`speed<>`**. O uso da biblioteca SIUnits permitiu ao compilador realizar a análise dimensional da função.

4.1.1. Problema

Para se implementar um sistema de análise dimensional baseado no sistema de tipo da linguagem, utilizando programação convencional, seria necessário criar um tipo de dados distinto para cada tipo de grandeza física, e sobrecarregar os operadores aritméticos para cada grandeza, de modo a criar inter-relações adequadas entre elas.

O Sistema Internacional de Unidades define “sete quantidades base (dimensões) mutuamente independentes: distância, massa, tempo, corrente elétrica, temperatura termodinâmica, quantidade de substância e intensidade luminosa” [Br01] e adicionalmente pode ser criado um número potencialmente infinito de unidades de medida derivadas pela composição dessas quantidades bases.

Com isso, para se criar um sistema de análise dimensional física baseado no sistema de tipos da linguagem, seria preciso que fossem criados infinitos tipos e “infinitos ao

quadrado" operadores sobrecarregados, o que é obviamente impossível de ser implementado manualmente.

4.1.2. Solução

Utilizando TMP é possível solucionar o problema da implementação da análise dimensional utilizando uma "Fábrica de Tipos". Por hora, considere somente duas grandezas físicas elementares, distância e tempo. Podemos criar uma base, que chamaremos de Base Dimensional, que represente os valores dos expoentes dimensionais dessas duas quantidades base e que possa conter o valor numérico da grandeza física em tempo de execução:

```
template<int D, int T> struct dim {
    enum { DIST=D, TIME=T };
    double value;
    dim(double x) : value(x) {};
};
```

E baseado nisso, podemos definir as unidades de medida para as grandezas físicas da seguinte forma:

```
typedef dim<0,0> scalar; // escalar (sem unidade)
typedef dim<1,0> distance; // distancia (metros ^ 1)
typedef dim<0,1> time; // tempo (segundo ^ 1)
typedef dim<0,-1> freq; // freqüência ( segundo ^ -1 )
typedef dim<1,-1> speed; // velocidade ( (metro ^ 1) * (segundo ^ -1) )
typedef dim<1,-2> acell; // aceleração ( (metro ^ 1) * (segundo ^ -2) )
```

Dessa forma, acabamos de atender ao primeiro requisito do nosso sistema de análise dimensional, criamos um número potencialmente infinito de tipos de dados. Agora precisamos criar operadores sobrecarregados de modo a manter a coerência entre as diversas unidades de medida:

```
template<int D, int T>
friend dim<D,T> operator+(const dim<D1,T1>& lhs, const dim<D2,T2>& rhs) {
    return(dim<D,T>(lhs.value + rhs.value));
};

template<int D1, int T1, int D2, int T2>
friend dim<(D1 + D2), (T1 + T2)>
    operator*(cosnt dim<D1,T1>& lhs, const dim<D2,T2>& rhs)
{
    return(dim<(D1 + D2), (T1 + T2)>(lhs.value + rhs.value));
};
```

Os operadores de subtração e divisão podem ser obtidos de maneira semelhante. A biblioteca SIUnits é baseada nos mesmos princípios definidos acima, porém são consideradas todas as sete quantidades base definidas na SI. É também usada a técnica de Listas de Tipos para armazenar a Base Dimensional reduzindo assim por sete o volume de código duplicado, pois as operações meta-programadas podem ser aplicadas recursivamente para toda a lista ao invés de realizadas individualmente elemento por elemento. Adicionalmente, também é utilizada a técnica de Meta-razão (*Ratio*), que possibilita representar Bases de Valor que contenham números racionais, e assim é possível utilizar tanto potências como raízes nas bases dimensionais.

5. Conclusão

Nesse trabalho foi apresentado um resumo dos conceitos básicos e do histórico da técnica de programação conhecida como meta-programação por templates. Foram apresentadas as técnicas mais elementares de TMP, através das quais foram mostrados os conceitos básicos do mecanismo de templates como interpretador para programação funcional.

Foi mostrado o conceito de Base como elemento fundamental de armazenamento e obtenção de dados e as Listas de Tipos como estrutura de dados. Foi mostrado o conceito geral de Meta-função e da especialização parcial de *templates* como forma de execução condicional em tempo de compilação, assim como idiomas que podem ser usados para implementar tipos específicos de Meta-funções. Foi ainda feito um estudo de caso analisando uma biblioteca de programação que faz uso de técnicas de TMP, de modo a apresentar uma amostra dos tipos de problemas que podem ser solucionados através do uso de técnicas de TMP.

A TMP não é, em geral uma técnica aplicável a uma família de problemas correlatos, mas a casos específicos de uma gama variada de problemas. Em geral sua aplicação é mais usual em bibliotecas de programação como forma de reduzir a tensão natural entre expressividade e desempenho.

Outro ponto importante é que apesar da sintaxe incomum, a TMP é fundamentalmente programação funcional, e uma vez superada a barreira de paradigma a aplicação de soluções meta-programadas passa a ser uma valiosa adição a caixa de ferramentas conceitual de um programador.

5.1. Trabalhos Futuros

A TMP ainda está na sua infância, e existem muitos aspectos dessa técnica de programação que podem ser explorados em trabalhos futuros.

Nesse trabalho foram documentadas as técnicas mais elementares e fundamentais da TMP, no entanto existem diversas outras, dentre as quais podemos destacar *Expression Templates* [Ve95a] usada em [BLIT] ou hierarquias de tipos usados em [LOKI] que servem como base para a criação de estruturas do tipo tupla [BTUL] [IS03], dentre outras. Um trabalho futuro pode ser dedicado a expandir o presente catálogo incluindo técnicas mais avançadas e específicas.

Além do trabalho pioneiro em [Ve95], tem havido um grupo reduzido de esforços no sentido de realizar a análise quantitativa das vantagens e desvantagens em termos da performance da aplicação de técnicas de TMP. Isso se deve principalmente a estreita dependência que existe entre as técnicas de TMP e a ferramenta de compilação utilizada. Tendo as técnicas de TMP sido devidamente catalogadas um possível trabalho futuro poderia avaliar os tempos de execução numa base de técnica por técnica, e eventualmente até mesmo uma análise comparativa entre compiladores.

Como foi dito na introdução a TMP surgiu quase que por acidente, e dessa forma muitos recursos que seriam de extrema utilidade para o desenvolvimento de meta-programas e técnicas de meta-programação não estão disponíveis na linguagem C++, simplesmente pelo fato que esses recursos não possuem aplicação direta para programas convencionais. Outro possível trabalho futuro consiste em investigar quais recursos úteis para TMP não estão disponíveis na linguagem C++ padrão, e qual a viabilidade desses recursos serem incluídos em uma ferramenta de compilação sem prejuízo para as funcionalidades já existentes na linguagem.

6. Bibliografia

[Br01] Walter Brown. Applied Template Metaprogramming in SIUNITS: The Library of Unit-Based Computation. 2001. disponível em www.wtf.ol.org/~Brown/brown.pdf, acesso em 2004/04/14.

[BLIT] Blitz++ Object-Oriented Scientific Computing. disponível em www.oonumerics.org/blitz acesso em 2004/04/14.

[BMPL] The Boost MPL Library. disponível em www.boost.org/libs/mpl, acesso em 2004/04/14.

[BTTL] The Boost Type Traits Library. disponível em www.boost.org/libs/type_traits, acesso em 2004/04/14.

[Cz00] Krzysztof Czarnecki, Ulrich Eisenecker. Generative Programming: Methods, Tools, and Applications. Addison-Wesley. 2000. ISBN: 0-201-30977-7.

[Ga00] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos. Bookman 2000. ISBN: 85-7307-610-0 .

[GNUC] GNU GCC: GNU Compiler Collection. disponível em www.gcc.gnu.org, acesso em 2004/04/14

[Gu02] Aleksey Gurtovoy, David Abrahams. The Boost C++ Metaprogramming Library. 2002. disponivel em www.boost.org/libs/mpl/doc/paper/mpl_paper.pdf, acesso em 2004/04/14.

[IS98] ISO/IEC 14882:1998(E), Programming Languages - C++. ISO/IEC. 1998.

[IS03] ISO/IEC. What will be in the Library TR?. ISO/IEC. 2003. disponível em http://www.open-std.org/JTC1/SC22/WG21/docs/library_technical_report.html, acesso em 2004/04/14

[LOKI] The Loki Library. disponivel em www.sourceforge.net/projects/loki-lib, acesso em 2004/04/14.

[Sh02] Tim Sheard, Simon Peyton Jones. Template Meta-Programming for Haskell. Haskell Workshop, Pittsburgh. 2002. disponivel em www.cse.ogi.edu/~sheard/papers/meta-haskell.ps acesso em 2004/04/14.

[Th99] Simon Thompson. Haskell: The Craft of Functional Programming. 2a ed. Addison-Wesley. 1999. ISBN: 0-201-34275-8.

[Un94] Erwin Unruh. Prime Number Computation: ANSI X3J16-94-0075/ISO WG21-462. 1994. disponivel em www.erwin-unruh.de/Prim.html acesso em 2004/04/14.

[Va02] David Vandevoorde, Nicolai Josuttis. C++ Templates: The Complete Guide. Addison-Wesley. 2002. ISBN: 0-201-73484-2.

[Ve95] Todd Veldhuizen. Using C++ template metaprograms. C++ Report Vol. 7 No. 4, pp. 36-43. 1995. disponível em www.osl.iu.edu/~tveldhui/papers/Template-Metaprograms/meta-art.html acesso em 2004/04/14

[Ve95a] Todd Veldhuizen. Expression Templates. C++ Report, Vol. 7 No. 5, pp. 26-31.1995. disponível em www.osl.iu.edu/~tveldhui/papers/Expression-Templates/exprtmpl.html acesso em 2004/04/14.

[Ve00] Todd Veldhuizen. Expresion Templates in Java. Generative and Component-Based Software Engineering, Erfurt, Germany. 2000. disponível em http://www.osl.iu.edu/~tveldhui/papers/2000/gcse00 acesso em 2004/04/14.

[Ve03] Todd Veldhuizen. C++ Templates are Turing Complete. 2003. disponível em www.osl.iu.edu/~tveldhui/papers/2003/turing.pdf acesso em 2004/04/14.

[ZOOM] Fermilab Physics Class Library Task Force (ZOOM). disponível em www.fnal.gov/docs/working-groups/fpcltf/Pkg/WebPages/zoomReal.html acesso em 2004/04/14.