Universidade Tecnológica Federal do Paraná – Câmpus Curitiba

**Relatório para Disciplina Linguagens e Compiladores PON**

Profs. João Alberto Fabro e Jean Marcelo Simão

*Disciplina conjunta entre PPGCA e CPGEI*

# Compilação C++ Estático

*StaticCPPCompiler.cpp*

Fernando Schutz e Adriano F. Ronszcka

fschutz@gmail.com, ronszcka@gmail.com

## *1. Introdução*

O projeto do compilador para C++ já existia nas versões anteriores do compilador PON, porém para a versão de C++ Estático (Static C++), o projeto foi reiniciado do zero, não aproveitando o que já havia sido feito. Tal decisão de projeto foi tomada justamente para não haver legados, bem como reforçar o aprendizado nas técnicas do PON.

## *2. Processo de construção*

Após reuniões, algumas ideias foram propostas:

- Uso de métodos e funções INLINE
  - Tais códigos são tratados como uma definição de macros para o compilador. Assim, a chamada da função é substituída pelo corpo (ou código-fonte) da função declarada.
  - É extremamente eficiente para funções pequenas (caso do PON que tem poucos códigos nativos), pois evita a geração de código para chamada e retorno da função.
  - Todas as funções e métodos são declarados e implementações no arquivo .h, ou seja, não há mais o arquivo específico .cpp contendo os códigos-fonte da implementação.
- Todos os elementos deveriam ser STATIC
  - Existe uma porção de armazenamento específica e singular para tais elementos, não importando sua quantidade. Todos os objetos compartilham este mesmo espaço de armazenamento. Assim, o compartilhamento de "informações" entre estes se torna mais próximo e, consequentemente, mais rápido.
- Cada entidade PON se tornou uma classe
  - Como agora tudo se tornaria estático, definiu-se que cada elemento PON seria uma classe específica.
  - Este processo não ficou natural, pois em implementações com muitos elementos, geram vários arquivos na mesma pasta, dificultando o processo de busca. Porém, visando facilitar qualquer monitoramento, a nomenclatura utilizada na nomeação dos arquivos buscou um padrão isonômico e intuitivo.

Assim, sobre a cadeia de notificação PON, quatro elementos foram construídos:

- Attributes: definiu-se que os Attributes teriam um atributo de valor e um método para ajustar seu valor (setValue()).
- Premise: possui um método que controla o processo de notificação, e um método que controla a inicialização da Premise. Uma questão importante foi realizar a cópia dos valores dos atributos durante a inicialização. Isso faz com que os valores possam ser comparados fidedignamente, sendo estes outros atributos ou valores.
- Subcondition: congrega os médotos para incrementar e decrementar a quantidade de Premises verdadeiras durante o processo de inferência.
- Methods: Neste não foi gerado apenas o arquivo .h mas também o arquivo .cpp, que contém a implementação do método.

A Figura 1 demonstra o fluxo de execução da cadeia de notificações PON em StaticCPP.

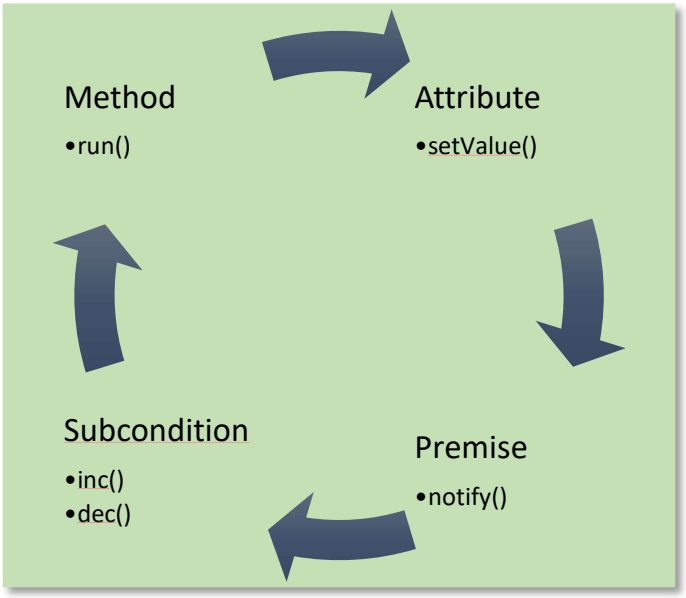


*Figura 1: Fluxo de execução da cadeia de notificações PON em StaticCPP*

O processo de criação, portanto, seguiu a ordem dos métodos predeterminados pelo Compilador existente. Em todos os métodos a árvore de elementos do PON já está devidamente construída, sendo necessário apenas o mapeamento destes para uma estrutura legível na linguagem C++. Assim, a seguinte sequencia foi utilização

- createAllRules(mapRules): neste método é feita a iteração entre todas as Rules do código PON. Assim é possível construir as *Conditions* e *Subconditions*. Dentro destas é feita uma varredura na *Instigations* relacionadas, visando inserir a chamada dos *Methods* relacionados a cada *Condition* no método que incrementa a quantidade de *Premises* verdadeiras. A Figura 2 apresenta um código-fonte de uma subcondition.
- createAllPremises(mapPremises): este método faz a iteração em todas as Premises e valida a qual conditions e subconditions que as mesmas pertencem para incluir em seu arquivo (#include). São criados os métodos de inicialização e comparação entre os valores dos atributos.
- createAllInstantiations(mapInstantiations): itera-se entre todas as Instantiations afim de localizar os métodos e os atributos relacionados. Também é feita a varreduras nas Premises para que se possa realizar a inclusão destas.

```
#pragma once
#include "event_mtReset.h"
#include "gate_mtClosing.h"

class subClosing{
    public: static int count;
            static inline void inc(){
               subClosing::count++;
               if (subClosing::count == 2){
                   event_mtReset::run();
                   gate_mtClosing::run();
               }
            }
            static inline void dec(){
               subClosing::count--;
            }
};
```

*Figura 2: Exemplo de Subcondition - "subClosing.h"*

```
#pragma once
#include "subOpening.h"
class prGateIsClosed{
    public: static bool state;
            static int cpy1st_att;
            static int cpy2nd_att;
            static inline void init(){
               prGateIsClosed::cpy1st_att = 0;
               prGateIsClosed::cpy2nd_att = 0;
               if (prGateIsClosed::cpy1st_att  ==  prGateIsClosed::cpy2nd_att){
                   prGateIsClosed::state = true;
                   subOpening::inc();
               }
            }
            static inline void compare(){
               if (prGateIsClosed::cpy1st_att  ==  prGateIsClosed::cpy2nd_att){
                 if (prGateIsClosed::state == false){
                    prGateIsClosed::state = true;
                    subOpening::inc();
                 }
               } else {
                 if (prGateIsClosed::state == true){
                    prGateIsClosed::state = false;
                    subOpening::dec();
                 }
               }
            }
            static inline void notify_gate_atGateState(int newValue){
                prGateIsClosed::cpy1st_att = newValue;
                prGateIsClosed::compare();
            }
};
```

*Figura 3: Exemplo de Premise – "prGateIsClosed.h"*

```
#pragma once
#include "prGateIsClosed.h"
#include "prGateIsOpened.h"

class gate_atGateState{
    public: static bool value;
            static inline bool setValue(bool newValue){
                if (gate_atGateState::value != newValue) {
                    gate_atGateState::value = newValue;
                    prGateIsClosed::notify_gate_atGateState(newValue);
                    prGateIsOpened::notify_gate_atGateState(newValue);
                }
            }
    };
```

*Figura 4: Exemplo de Attribute - "gate_atGateState.h"*

```
#pragma once

class gate_mtClosing{
    public: static bool run();
};
```

(a)

```
#include "gate_mtClosing.h"
#include "gate_atGateState.h"

inline bool gate_mtClosing::run(){
    gate_atGateState::setValue(0);

}
```

(b)

*Figura 5: Exemplo de Method – (a)"gate_mtClosing.h" e (b) "gate_mtClosing.cpp"*

No programa principal (main.cpp) devem ser feitas as inicializações de todos os atributos estáticos, bem como a execução dos métodos de inicialização das *Premises*. A Figura 6 apresenta um trecho de código exemplificando o processo de inclusão de todos os elementos PON, bem como a inicialização de seus atritubos.

```
#include "subClosing.h"
#include "subOpening.h"
#include "prGateIsClosed.h"
#include "prGateIsOpened.h"
#include "prRemoteControlOn.h"
#include "event_mtReset.h"
#include "event_mtReset.cpp"
#include "event_atEventState.h"
#include "gate_mtOpening.h"
#include "gate_mtOpening.cpp"
#include "gate_mtClosing.h"
#include "gate_mtClosing.cpp"
#include "gate_atGateState.h"

int subClosing::count = 0;
int subOpening::count = 0;
bool prGateIsClosed::state = false;
int prGateIsClosed::cpy1st_att = 0;
int prGateIsClosed::cpy2nd_att = 0;
bool prGateIsOpened::state = false;
int prGateIsOpened::cpy1st_att = 0;
int prGateIsOpened::cpy2nd_att = 0;
bool prRemoteControlOn::state = false;
int prRemoteControlOn::cpy1st_att = 0;
int prRemoteControlOn::cpy2nd_att = 0;
bool event_atEventState::value = false;
bool gate_atGateState::value = false;
```

*Figura 6: Snapshot de código-fonte do programa principal - "main.cpp"*

Testes foram efetuados utilizando o exemplo do portão eletrônico. Em uma máquina comum do laboratório L204 configurou-se o sistema operacional para trabalhar no modo monotarefa e monousuário, minimizando assim a influência do Sistema Operacional sobre o processo de testes.

O sistema a ser testado deveria fazer 1.000.000 (um milhão) de mudanças do portão de aberto – fechado – aberto. Os resultados foram:

- Na compilação normal (sem otimização):
    - PON em linguagem C: 32000 milissegundos
    - PON em linguagem C++: 82000 milissegundos
    - PON em linguagem C++ estática: 52000 milissegundos

Os primeiros resultados atingiram o esperado, ou seja, possivelmente não teria uma performance melhor que a linguagem C “pura” porém melhor que a linguagem C++ não estática. Porém um dos objetivos do projeto era o teste com a compilação otimizada, utilizando-se o parâmetro –O3 durante a compilação. Os resultados foram:

- Na compilação otimizada (-O3):

- PON em linguagem C: 21000 milissegundos
- PON em linguagem C++: 19000 milissegundos
- PON em linguagem C++ estática: 9000 milissegundos

Ou seja, na versão com otimização de código, a nova versão proposta era de duas a três vezes mais rápida que as versões existentes de compilação do código PON. A Figura 7 apresenta tais resultados.

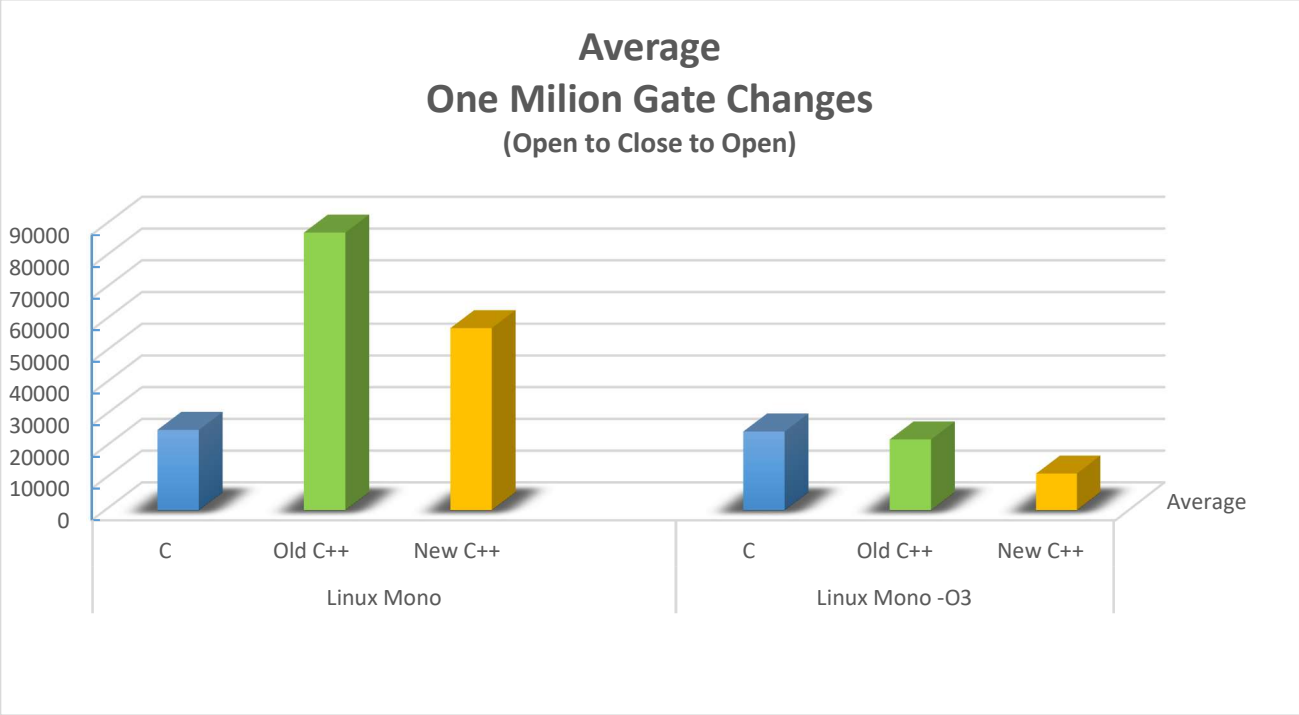


*Figura 7: Resultado dos testes*

Assim, pode-se concluir que a utilização de construções de códigos INLINE e elementos STATIC, juntamente com a compilação otimizada, resultou num código veloz e eficiente.

## 3. *Modificações em 2016*

Após a utilização do sistema StaticCPP em problemas mais complexos do que o portão eletrônico, tradicionalmente implementado nas disciplinas, percebeu-se algumas falhas no código, bem como a falta de recursos para especificar situações pertinentes ao PON. As seguintes seções visam detalhar tais modificações.

### 3.1. Valor inicial de Attributes e Premises

Na primeira versão do C++ estático, ao se iniciar as *Premises*, era feita a verificação de sua validação, ou seja, caso o valor padrão se encaixasse expressão lógico-causal tornando a *Premise* verdadeira, o valor desta se tornava verdadeiro, e contava na validação da Subcondition e na *Rule*. Este processo foi retirado, pois estava causando problemas na lógica de alguns algoritmos, pois fazia com que, ao se ajustar o valor padrão inicial, o *Method* relacionado à Action era disparado, invalidando a cadeia de notificações inicial.

O programador é forçado, agora, a tecer a lógica do algoritmo de acordo com o que deve ser ou não notificado ao ajustar os valores iniciais dos *Attributes*. Isso faz com que o desenvolvedor tenha consciência da lógica aplicada à cadeia de notificações.

```
407     hidneu0_attDone::setValue(false, _NOPAdds::RENOTIFY);
408     hidneu1_attDone::setValue(false, _NOPAdds::RENOTIFY);
409     hidneu2_attDone::setValue(false, _NOPAdds::RENOTIFY);
410     hidneu3_attDone::setValue(false, _NOPAdds::RENOTIFY);
411     hidneu4_attDone::setValue(false, _NOPAdds::RENOTIFY);
412     outneu0_attDone::setValue(false, _NOPAdds::RENOTIFY);
413     outneu1_attDone::setValue(false, _NOPAdds::RENOTIFY);
414     outneu2_attDone::setValue(false, _NOPAdds::RENOTIFY);
415
416     net_attReadTrainInputs::setValue(0, _NOPAdds::NO_NOTIFY);
417     net_attTotTrainInputs::setValue(1, _NOPAdds::NO_NOTIFY);
```

No exemplo acima há o ajuste inicial de oito *Attributes* booleanos com valores false, sendo que tal valor já deve ser contado na *Rule* relacionada (RENOTIFY). Posteriormente, nas linhas 416 e 417, dois *Attributes* inteiros são valorados, porém tal processo não deve ser levado em consideração pela *Rule*, ou seja, não será avaliado na *Premise* durante sua atribuição.

## 3.2. Notificação de Atributos

A valoração de *Attributes* é realizada por meio do método *setValue*, que passa agora a ter dois parâmetros: valor e tipo de notificação.

### 3.2.1. Tipo de Notificação

- _NOPAdds::STANDARD: comportamento padrão de notificações, ou seja, caso haja mudança no valor é notificado, caso contrário não há notificação. Caso seja deixado em branco, o algoritmo assume tal comportamento.
- _NOPAdds::RENOTIFY: faz com que haja notificação mesmo sem a mudança no valor do atributo.
- _NOPAdds::NO_NOTIFY: não há notificação mesmo com a mudança no valor do atributo.

```
outneu1_attDone::setValue(false, _NOPAdds::RENOTIFY);
outneu2_attDone::setValue(false, _NOPAdds::RENOTIFY);

net_attReadTrainInputs::setValue(0, _NOPAdds::NO_NOTIFY);
net_attTotTrainInputs::setValue(1, _NOPAdds::NO_NOTIFY);
```

## 3.3. Interpretação de Métodos

Todo método declarado na FBE deve ser escrito em uma linha, sendo que cada linha de comando deve ser separada por ponto e vírgula (;). Assim, optava-se por escrever os métodos após a compilação de tradução do algoritmo PON para a linguagem alvo. O grande problema é quando tem-se Regras de Formação, pois na pré-compilação cada uma das form*Rule*s será dividida em várias, de acordo com a quantidade de instancias da mesma. Assim, algumas novas interpretações foram realizadas nos métodos.

### 3.3.1. Inclusão de bibliotecas

Caso o programador necessite de alguma biblioteca, esta poderá ser declarada no início do corpo do método.

```
method mthNetStart() begin_method  #include <iostream>
<def>using namespace std;</def> cout << "mth Net Start" << endl;
 attClock = 1<N>; attReadTrainInputs = 0<R>;  end_method
```

### 3.3.2. Comandos e definições

Caso seja necessário inserir uma configuração ou definição que deva aparecer antes do corpo do método e depois das *includes* iniciais da classe, basta colocar o código entre as palavras-chave <def> e </def>

```
method mthNetStart() begin_method  #include <iostream>
<def>using namespace std;</def> cout << "mth Net Start" << endl;
 attClock = 1<N>; attReadTrainInputs = 0<R>;  end_method
```

### 3.4. Tipo de Atribuição/Notificação

Logo após a valoração do atributo, insere-se um valor entre os caracteres <> para definir o tipo de notificação:

- <S>: _NOPAdds::STANDARD
- <R>: _NOPAdds::RENOTIFY
- <N>: _NOPAdds::NO-NOTIFY

```
method mthNetStart() begin_method  #include <iostream>
<def>using namespace std;</def> cout << "mth Net Start" << endl;
 attClock = 1<N>; attReadTrainInputs = 0<R>;  end_method
```

### 3.5. Adequação no ajuste (set) e obtenção dos valores de Attributes

Por padrão, a compilação insere os comandos para a modificação e obtenção dos valores dos Attrbitues, conforme a sintaxe padrão para o C++ Estático. O método da figura abaixo é resultante da compilação do método PON utilizado nos exemplos anteriores.

```
1  #include "net_mthNetStart.h"
2  #include "_NOPAdds.h"
3  #include "net_attClock.h"
4  #include "net_attReadTrainInputs.h"
5  #include <iostream>
6
7  using namespace std;
8  inline bool net_mthNetStart::run(){
9   cout << "mth Net Start" << endl;
10      net_attClock::setValue(1, _NOPAdds::NO_NOTIFY);
11      net_attReadTrainInputs::setValue(0, _NOPAdds::RENOTIFY);
12 }
```

## *4. Classe padrão _NOPAdds.h*

A cada compilação, será criada uma classe padrão a ser utilizada em todas as classes geradas na compilação. Até o presente, tal classe possui apenas a definição das constantes utilizadas na valoração de Attributes em seus tipos, conforme apresentado anteriormente.

```
#ifndef _NOPADDS_H_
#define _NOPADDS_H_

class _NOPAdds {
    public: const static int STANDARD = 0;
            const static int RENOTIFY = 1;
            const static int NO_NOTIFY = 2;
    };
#endif
```

## *Referências*


RONSZCKA, A. F. **Contribuição para a concepção de aplicações no paradigma orientado a notificações (PON) sob o viés de padrões**, 2012. UTFPR.

FERREIRA, C. A. **Linguagem e compilador para o Paradigma Orientado a Notificações (PON): avanços e comparações**, 2015. UTFPR.

VALENÇA, G. Z.; BANASZEWSKI, R. F.; RONSZCKA, A. F.; et al. Framework PON, Avanços e Comparações. III Simpósio de Computação Aplicada. **Anais...** , 2011. Passo Fundo/RS.

BANASZEWSKI, R. F. **Paradigma Orientado a Notificações: Avanços e Comparações**, Dissertação de Mestrado (CPGEI/UTFR). 2009. Curitiba/PR: UTFPR.