

---

Processamento de Linguagens  
**Parte I - B**  
FLINT  
Flex Inline Template

---

Bruno Cancelinha  
(A75428)

José Bastos  
(AXXXXX)

Marcelo Miranda  
(AXXXXX)

15 de Abril de 2017



**Universidade do Minho**

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>1</b>
1.1	Estrutura do relatório . . . . .	1
<b>2</b>	<b>Análise e especificação</b>	<b>2</b>
2.1	Análise do problema . . . . .	2
2.2	Objetivos . . . . .	2
<b>3</b>	<b>Ferramenta FLINT</b>	<b>3</b>
3.1	Notação . . . . .	4
3.1.1	Funções . . . . .	4
3.1.2	Declaração de variáveis . . . . .	4
3.1.3	Variáveis em linha . . . . .	5
3.1.4	Mapeamentos . . . . .	5
3.1.5	Expressões condicionais . . . . .	5
3.1.6	Chamamento de funções . . . . .	6
3.2	Output . . . . .	6
3.3	Flags . . . . .	7
<b>4</b>	<b>Implementação</b>	<b>8</b>
<b>5</b>	<b>Codificação e Testes</b>	<b>9</b>

<b>6</b>	<b>Conclusão</b>	<b>10</b>
<b>A</b>	<b>Exemplos</b>	<b>11</b>

## **Resumo**

FLINT - Uma ferramenta simples que visa a tornar mais fácil a criação de templates para strings em C.

# Capítulo 1

## Introdução

### 1.1 Estrutura do relatório

# Capítulo 2

## Análise e especificação

### 2.1 Análise do problema

Em 1972 estava Dennis Ritchie a terminar o que se veio a tornar uma das mais importantes linguagens de programação, a linguagem C. Devido à sua idade e filosofia como linguagem imperativa de baixo nível, *strings* não são suportadas naturalmente ou, melhor dizendo, não são suficientemente flexíveis para tratar, como é muitas vezes o caso, linguagens naturais.

É muito comum querer ter uma *string* com conteúdo variável, um simples cumprimento como "Olá NOME" em que NOME representa uma outra *string* com o nome do recetor. Para este efeito é frequentemente usada a função `sprintf` da biblioteca `<string.h>` que permite escrever vários dados numa *string* com um formato específico. Esta solução funciona e de facto é eficaz, mas é tediosa quando se pretende fazer a um conjunto significativo de linhas. E, devido à natureza estática de *arrays* em C, não é fácil garantir que não se excede o limite máximo da *string*.

### 2.2 Objetivos

Seria ideal imaginarmos uma ferramenta que simplificasse todo este processo de modelação e que garantisse uma maneira eficaz de construir um numeroso conjunto de *strings* formatadas com dados variáveis. Essa ferramenta teria de ser capaz de transformar um modelo, em código C compilável e eficiente. Teria que usar *strings dinâmicas*, mantendo sempre atenção para não permitir *leaks* de memória. Estamos portanto a falar de um **processador de modelos**, como será chamado em Português, ou *template engine* na língua anglo-saxónica, em par com uma **linguagem de modelação**, ou *template language*.

É com este objetivo que desenvolvemos uma *linguagem de modelação* e o seu correspondente *processador de modelos*, que recebe o modelo especificado nessa linguagem e o converte em código C válido.

# Capítulo 3

## Ferramenta FLINT

A ferramenta **FLINT** (**F**lex **I**ncode **T**emplate), vem como resposta ao problema de tratamento de *strings* em C, facilitando a criação e uso de modelos na linguagem. Traz consigo um processador de modelos, escrito usando **Flex**, que converte a linguagem de modelação **FLINT** numa função em C que recebe os dados usados durante o modelo, e retorna a *string* já formatada. FLINT tem a capacidade de reconhecer, na maior parte dos casos, as variáveis usadas no modelo, essas variáveis aparecerão como argumentos da função gerada pela ordem que aparecerem no código, ignorando casos em que aparecem repetidas.

Um documento começa com a primeira linha `//FLINT` para não deixar dúvidas de que se trata de facto de um ficheiro FLINT. De seguida vem as funções FLINT que será explicada em mais detalhe da secção seguinte mas, basicamente, funções FLINT são um *template*, são a parte de texto que será convertida em código C. O FLINT é suficientemente inteligente para distinguir o código C dos modelos, portanto mantém intacto qualquer código que o ficheiro original contenha.

As *strings* em C, como já foi referido neste relatório, não estão pensadas para lidar com tamanhos variáveis, isto é, não são dinâmicas. Não passam de uma sequência fixa de bytes terminada com um byte a NULL, o chamado *NULL terminator*. Embora este comportamento seja compreensível e, muitas vezes adequado, para um processador de modelos é esperado que seja capaz de suportar qualquer que seja o tamanho dos dados que lhe desejamos passar, e não podemos prever qual o tamanho dos dados que serão utilizados em conjunto com o modelo. Por esta mesma razão, decidimos que seria necessário o uso das **GStrings** do **GLib**. Assim, o código gerado pelo **FLINT** usará **GStrings**. O FLINT certifica-se que o código de *output* tem o `#include <glib.h>`, colocando-o caso este não esteja presente no ficheiro de *input* e não fazendo nada se estiver.

É possível converter algo como:

---

```
1 Greeting={{Bem-vindo [% nome %]}.}}
```

---

No código C seguinte:

---

```
1 char* Greeting( char* nome) {
2     GString *str = g_string_new(NULL);
3
4     g_string_append_printf(str, "Bem-vindo %s.", nome);
5
6     return g_string_free(str, FALSE);
7 }
```

---

## 3.1 Notação

Nesta secção vamos ver como é a notação da linguagem de modelação FLINT e a sua robustez. Para distinguir caracteres pertencentes ao modelo de variáveis e outras características da linguagem FLINT, estas palavras especiais são colocadas entre [% %], sendo que qualquer caracter que não pertença a esta notação FLINT, incluindo \n e outros caracteres especiais, serão considerados na string a retornar.

### 3.1.1 Funções

Como já foi dito acima, FLINT baseia-se em funções que são nada mais que modelos a serem convertidos em C.

Uma função FLINT é definida entre chavetas duplas e é-lhe atribuída um nome com o símbolo de igual. Assim, Func={ { MODELO } } representa uma função de nome Func, neste caso MODELO é só um marcador de posição sem qualquer valor semântico. Devido ao facto de qualquer caracter dentro das duplas chavetas será considerado na *string* resultante, a posição das duplas chavetas } } é bastante relevante.

### 3.1.2 Declaração de variáveis

Apesar da capacidade do FLINT, de automaticamente detetar variáveis no modelo e as coloca como atributos da função gerada, é por vezes necessário explicitar essas variáveis. Assim, para declarar uma variável com o nome `idade` do tipo `int`:

---

```
1 [% VAR int idade %]
```

---

Se o nome da variável já estiver a ser utilizado antes da declaração, o FLINT já a terá registado, embora é provável que com um tipo diferente do pretendido e esta declaração é simplesmente ignorada. De qualquer maneira, esta funcionalidade é útil em IFs e em *variáveis em linha*, como veremos à frente.



### 3.1.3 Variáveis em linha

O mais simples dos processadores de modelos tem que incluir, pelo menos, a funcionalidade de substituição simples de variáveis no meio do *template*. Assim, sendo esta uma característica tão simples, será simplesmente o nome da variável entre a notação do FLINT [% %], como será possível confirmar no exemplo dado no início deste capítulo. Quando o FLINT encontra **variáveis em linha**, ou *inline variables*, regista automaticamente o seu nome para ser colocada nos argumentos da função gerada, por omissão regista a variável com tipo `char*`, mas o FLINT suporta o caso da variável já ter sido declarada com o tipo `float` ou `int`. Assim é possível fazer algo como:

---

```
1 Greeting={{Olá [% nome %]}!  
2 [% VAR int idade %]  
3 Tu tens [% idade %], certo? }}
```

---

Neste exemplo, FLINT regista a variável `char* nome` e `int idade`, por esta ordem.

### 3.1.4 Mapeamentos

Para além da trivial variável em linha, o FLINT suporta **mapeamentos**, que são representados da seguinte maneira:

---

```
1 [% MAP Func n 1 %]
```

---

Mapeamentos permitem mapear a função `Func` a uma lista `1` com `n` elementos. Chamando a função `Func` para cada elemento de `1`, havendo a restrição de que a função apenas pode receber um argumento do tipo `char*`.

Quando o FLINT encontra um mapeamento, regista a variável `n` do tipo `int` e `1` do tipo `char**`.

### 3.1.5 Expressões condicionais

Nem sempre tudo o que está no modelo corresponde ao que desejamos apresentar. Podemos querer ter um modelo que varia consoante os dados. Trata-se aqui a questão de controlo de fluxo, permitir que certas partes do modelo sejam apresentadas ou não, consoante os dados perante a condição imposta.

O FLINT também permite **expressões condicionais** do tipo *if*, *else if* e *else*. Um bloco `IF` termina obrigatoriamente com `ENDIF` e pode, ou não, conter blocos `ELIF`, que representa o *else if*, e `ELSE`. Tanto o `IF` como o `ELIF` recebem uma condição como argumentos, qualquer variável encontrada nessa condição é registada como sendo do tipo `int`, a não ser que já tenha sido registada anteriormente.

---

```
1 [% IF sexo == FEMININO %]
2 Estás muito bonita hoje.
3 [% ELIF sexo == MASCULINO %]
4 Estás muito bonito hoje.
5 [% ELSE %]
6 Bom dia.
7 [% ENDIF %]
```

---

No exemplo acima, caso `sexo` seja `FEMININO`, é apresentada a *string* "Estás muito bonita hoje", "Estás muito bonito hoje" caso `sexo` seja `MASCULINO` ou "Bom dia." em qualquer outro caso. O FLINT automaticamente deteta a palavra `sexo` como uma variável do tipo `int`, e ignora as palavras `FEMININO` e `MASCULINO` assumindo que se tratam de constantes pelo facto de estarem escritas todas em maiúsculas. O FLINT vai detetar qualquer palavra minúscula como uma variável a não ser que esteja protegida pelo carácter `:`, muito útil quando a condição chama funções em C.

---

```
1 Olá [% nome %]!
2 [% IF (:strcmp:(nome, "Maria") == 0) || (:strcmp:(nome, "João") == 0) %]
3 Tu outra vez!?
4 [% ENDIF %]
```

---

Aproveitamos o exemplo de cima para demonstrar, para além do uso do *escape character* `:`, o uso de uma disjunção, embora conjunções também sejam possíveis, aliás, qualquer condição que funcionaria em C, irá funcionar no FLINT.

### 3.1.6 Chamamento de funções

É possível chamar funções FLINT dentro dum modelo usando a expressão `CALL`, seguida do nome da função e os seus argumentos. Estes argumentos são variáveis que constarão nos argumentos da função gerada, se não tiverem já sido referidos antes da expressão `CALL`, o FLINT vai assumi-los como `char*`. Para além da função `CALL`, o FLINT permite `BREAK` que sai do modelo.

---

```
1 Nome={{
2 Nome: [% nome %]}}
3
4 Idade={{
5 [% IF i < 18 %]
6 [% BREAK %]
7 [% ENDIF %]
8 Idade: [% i %]}}
9
10 Info={{
11 [% CALL Nome nome %]
12 [% VAR int idade %]
13 [% CALL Idade idade %]
14 }}
```

---

No exemplo acima, a função `Idade` termina logo se o valor de `i` for inferior a 18. Podemos também observar que a função `Info` chama as duas funções anteriores, tendo o cuidado de declarar a variável `idade` como um inteiro.

## 3.2 Output

---

```
1 char* FUNCAO( ... ) {
2     GString *str = g_string_new(NULL);
3
4     (...)
5     g_string_append_printf(str, "%s", VAR);
6     (...)
7
8     return g_string_free(str, FALSE);
9 }
```

---

## 3.3 Flags

# Capítulo 4

## Implementação

# Capítulo 5

## Codificação e Testes

# Capítulo 6

## Conclusão

- mais flags (customização) - controlo de erros - argumentos automaticos para CALL

# Apêndice A

## Exemplos