Universidade Federal do Rio Grande do Norte
Prática de Algoritmos e Estrutura de Dados I

Rivaldo Rodrigues

DOXYGEN

1 - Introdução

Esse tutorial tem como objetivo apresentar e estimular o aluno a usar uma ferramenta de documentação de código.

Em termo de re-usabilidade de software a documentação se mostra de vital importância visto que uma das principais funções de uma boa documentação de código é apresentar ao usuário uma visão geral dos programas ou da API (Application Programming Interface) a ser utilizado pelo código cliente sem, no entanto, expor detalhes da implementação, tornando assim mais fácil a utilização (re-utilização) do mesmo.

A utilização de uma ferramenta de documentação automática tem a vantagem de permitir que nos concentremos em elaborar uma boa descrição do código, ao invés de perdermos tempo com detalhes relativos à aparência que tal documentação terá para o usuário final.

Isso acontece pois, a principal função da ferramenta é utilizar as **marcações** geradas no código fonte para criar e organizar a documentação em formatos populares e publicações eletrônicas como **HTML** ou **pdf** (via latex).

Apresentaremos durante este texto a ferramenta Doxygen que consiste num sistema de documentação para C++, C, Java, Objective-C, Python, IDL e algumas extensões de PHP, C# e D. Versões do Doxygen podem ser encontradas para Windows, Linux e Mac OS.

Vale salientar que este documento buscar apresentar apenas os elementos básicos para se criar uma documentação utilizando Doxygen. Caso seja do interesse do leitor aprender a utilizar as funções avançadas de documentação, recomenda-se a leitura do tutorial do desenvolvedor em http://www.stack.nl/~dimitri/doxygen/manual.html

2 Instalação e configuração

Baixe gratuitamente a versão do Doxygen especifica para o seu sistema operacional no site www.doxygen.org ou de algum gerenciador de de pacotes (como o Synaptc no Linux).

Siga a procedimento de instalação de acordo com o seu sistema operacional.

Agora que o doxygen esta devidamente instalado, podemos utiliza-lo para criar documentação de arquivos. Para isso usamos o comando

```
$ doxygen arquivo.h (ou .cpp)
```

no mesmo diretório do arquivo. Após isso será criado duas pastas, uma com a documentação em HTML e outra em Latex.

Note portanto que o doxygen, por padrão, não criará a documentação que não sejam de classe, como, por exemplo, um arquivo cpp contendo só a main().

Mostraremos agora como configurar algumas opções básicas.

Alguns dos procedimentos para demonstrados podem ter uma sintaxe diferente dependendo da versão utilizada. Para configurar as opções ao se gerar o código deve se primeiramente criar um arquivo de configuração. Para isso digitamos o comando:

```
$ doxygen -g nome_do_arquivo
```

Com isso, o doxygen gerará um arquivo com o nome especificado contendo as opções de configuração. Uma leitura do arquivo de configuração nos permite conhecer informações sobre o que cada opção representa.

Algumas destas opções são:

OUTPUT_LANGUAGE = Brazilian
Gera o documento em português(Brasil)

EXTRACT_ALL = YES
Gera a documentação de todos os elementos
(incluindo funções fora de uma classe)

Estas e várias outras opções podem ser modificadas.

Para criar uma documentação com base no arquivo de configuração previamente gerado usamos o comando:

```
$ doxygen arquivo_conf  arquivo.h (ou cpp)
```

3 - Documentando o código

Um bloco de documentação no estilo Doxygen difere ligeiramente do padrão do C++, por exemplo, por requerer algumas **marcas** adicionais. Essas marcas difere da sintaxe do Doxygen pois este consiste em algumas ão responsáveis por permitir que o Doxygen reconheça que aquela parte do documento deve ser usada na hora de gerar a documentação.

Para cada item do código há dois tipos de descrição que juntos formam a documentação: **descrição breve** e **descrição detalhada**.

Ambas são opcionais porém, utilizar mais de uma descrição do mesmo tipo não é permitido.

Uma descrição breve consiste num comentário de única linha, já a descrição detalhada, como esperado, pode ocupar diversas linhas.

3.1 - Comentários detalhados

Há vários modos de se marcar uma descrição detalhada:

1. Pode ser usado o estilo do JavaDoc , o qual consiste em bloco de comentário estilo C iniciando-se com dois asteriscos (*):

```
/**
 * ... texto ...
 */
```

2. Alternativamente, você pode usar o estilo Qt e adicionar uma exclamação após o inicio do com bloco de comentário estilo C como mostra o exemplo:

```
/*!
 * ... texto ...
 */
```

   Em ambos os casos os asteriscos (*) das demais linhas, excerto o que finaliza o comentário (*/) são opcionais.

```
/*!
 * ... texto ...
 */
```

3. A terceira alternativa consiste em utilizar um bloco de pelo menos dois comentários de linha estilo C++ acrescentando uma barra adicional ou um sinal de exclamação.

```
///
/// ... texto ...
///

//!
//! ... texto ...
//!
```

4. Pode se também deixa o comentário mais visível como a exemplo de:

```
//////////////////////////////////////////////
/// ... texto ...
//////////////////////////////////////////////
```

3.2 - Comentários breves

Para se construir um comentário breve podemos:

1. Usar o comando **\brief** em um dos blocos de comentário. Esse comando termina após o fim do parágrafo, ou seja, após uma linha em branco.

```
/*! \brief Descrição breve
 *         continuação da descrição.
 *
 *  Inicio a descrição detalhada.
 */
```

2. Se a opção JAVADOC_AUTOBRIEF no arquivo de configuração estiver selecionado como **YES** pode ser usado o estilo JavaDoc, assim, o comentário vai iniciar automaticamente com um comentário breve o qual termina após o primeiro ponto final:

```
/** Descrição breve termina após este ponto. Segue então a
 *  descrição detalhada.
 */
```

Obtemos o mesmo efeito utilizando o estilo C++:

```
/// Descrição breve termina após este ponto. Segue então a
/// descrição detalhada.
```

3. Uma terceira forma é utilizar o estilo C++ de comentário de forma a não se estender por mais de uma linha. Como nos dois exemplos a seguir:

```
/// Descrição breve.
/** Descrição detalhada. */
```

ou

```
//! Descrição breve.
```

```
//! Inicio da descrição
//! detalhada.
```

Note que foi utilizado uma linha em branco no ultimo exemplo para separar os dois tipos de descrição.

O código seguinte não representa uma descrição valida, pois só é possível uma descrição breve por função, classe ou variável:

```
//! Descrição breve, a qual utilize mais
//! De uma linha é tida como uma descrição detalhada.
/*! Oops, Aqui temos outra descrição detalhada!
 */
```

Caso exista uma descrição breve antes de uma declaração e outra após a mesma, apenas a descrição que precede a declaração será utilizada. O mesmo ocorre para descrições detalhadas.

Aqui esta um exemplo de documentação no estilo Qt de uma classe em C++, mais adiante serão apresentadas outras ferramentas para descrevemos completamente a classe teste:

```
//!  Uma classe teste.
/*!
 *   Uma descrição mais elaborada.
 */

class Test
{
  Public:

    //! Construtor.
    /*!
        Uma descrição detalhada do construtor.
    */
    Test();

    //! Destrutor.
    /*!
        Uma descrição detalhada do desctrutor.
    */
   ~Test();

.......

};
```

Aqui esta a mesma classe documentada usando o estilo JavaDoc e a opção JAVADOC_AUTOBRIEF selecionada como YES:

```
/**
 *  Uma classe teste. Uma descrição mais elaborada.
 */

class Test
{
  public:

      /**
       * Construtor.
       * Uma descrição detalhada do construtor.
       */
      Test();

      /**
```

```
         * Destrutor
         * Uma descrição detalhada do destrutor.
         */
        ~Test();

    .. ...

    };
```

Note que embora o construtor e o destrutor não estejam implementados a documentação precede os mesmo, ou seja, a documentação vem onde o método, variável ou classe foi definido primeiro.

3.3 Documentação após a declaração de membros.

Caso você queira documentar os membros de dados, arquivos, estruturas, classes ou enumerações você terá que colocar a documentação antes do mesmo. No entanto, as vezes é mais viável escrever a documentação após a declaração. Para esse propósito nós usamos o símbolo < para relacionar o comentário.

Aqui nós temos alguns exemplos:

```
int var; /*!< Descrição detalhada após membro de dados */
```

ou:

```
int var; /**< Descrição detalhada após membro de dados */
```

ou

```
int var; //!< Descrição detalhada após membro de dados
         //!<
```

ou ainda:

```
int var; ///< Descrição detalhada após membro de dados
         ///<
```

Geralmente usa-se apenas uma descrição breve após os membros como segue:

```
int var; //!< Descrição breve de membros de dados
```

ou

```
int var; ///< Descrição breve de membros de dados
```

Vale salientar que este novo método de descrição segue as mesmas funcionalidade dos métodos anteriores.

No exemplo abaixo, temos o uso deste novos métodos:

```
/*! Uma classe teste */

class Test
{
public:

    /**! Uma enumeração.
     *   O bloco de documentação não pode ser colocado depois da
     *  enumeração */
    enum EnumType
    {
      int EVal1,     /**< Valor 1 */
      int EVal2      /**< Valor 2 */
};

    void member();   //!< Uma função membro

  protected:
    int value;       /*!< Um valor inteiro */
};
```

Nota:

Esses método só pode ser usado para documentar membros de dados e funções. Ele não pode ser usado para documentar um arquivo, classe, união, enumeração ou namespace.

3.4 - Documentando funções

Vários parâmetros adicionais podem ser acrescentados à documentação já apresentada no que se refere as funções. O quadro abaixo apresenta alguma delas:

\param Indica os parâmetros recebidos pela função.

\sa Caso se queira fazer referencia a outra função já existente, após a documentação ser gerada teremos um link para as funções referenciadas.

\return Indica o retorno da função

Abaixo temos exemplos de uso desses parâmetros na documentação de funções.

```
//! Uma função membro para teste.
/*!
    \param a um inteiro que representa o tamanho do array.
    \param s um ponteiro constante para um array de char.
    \return um inteiro.
    \sa Test(), ~Test(), testMeToo() and publicVar()
 */
int testMe(int a, const char *s);
```

ou podemos ainda utilizar o estilo JavaDoc

```
/**
 * Uma função membro para teste.
 * @param um inteiro que representa o tamanho do array.
 * @param s um ponteiro constante para um array de char.
 * @see Test()
 * @see ~Test()
 * @see testMeToo()
 * @see publicVar()
 * @return um inteiro.
 */
   int testMe(int a,const char *s);
```

3.5 - Dados de cabeçalho

O Doxygen possui alguns parâmetros específicos na criação de cabeçalhos em programas, tais cabeçalhos tem o intuito de informar dados como: o autor do arquivo, data da implementação, versão do arquivo, etc. Apresentaremos agora os principais parâmetros para um cabeçalho.

Utilizando um bloco de comentário como mostrado anteriormente, acrescentamos parâmetros especifico para cada utilidade. São eles:

\author Indica o autor da classe (arquivo)
\since Indica a data da primeira implementação
\version Indica a versão atual

A segui temos um exemplo de como esses parâmetros podem ser colocados em pratica utilizando o estilo Qt.

```
//!  Descrição do arquivo.
/*!
 *   \author Nome do Autor.
 *   \since 10/10/10
 *   \version 1.0
 */
```

O estilo JavaDoc também pode ser utilizado, como no exemplo abaixo:

```
/**  Descrição do arquivo.
/*
 *   @author Nome do Autor.
 *   @since 10/10/10
 *   @version 1.0
 */
```

4 - Exemplo

Por fim, apresentaremos um exemplo da documentação completa, a qual foi construída ao longo deste texto em estilo Qt (lembrando que a mesma pode ser construída no estilo JavaDoc).

Sugiro ao leitor que copie (ou escreva) o código abaixo em um arquivo (arquivo.h por exemplo) e use a ferramenta Doxygen para conferir o resultado gerado.

```
//!  Uma classe teste.
/*!  Uma descrição mais elaborada.
 *   \author Nome do Autor.
 *   \since 10/10/10
 *   \version 1.0
 */

class Test
{
  public:

    //! Construtor.
    /*!
      Uma descrição detalhada do construtor.
*/
    Test();

    //! Destrutor.
    /*!
      Uma descrição detalhada do desctrutor.
*/
   ~Test();

    //! Enumeração.
    /*! Uma descrição detalhada da enumeração */
    enum TEnum {
                 TVal1, /*!< Valor TVal1. */
                 TVal2, /*!< Valor TVal2. */
                 TVal3  /*!< Valor TVal3. */
               }
         //! Ponteiro para enumeração.
         /*! Detalhes */
         *enumPtr,
         //! Variável para enumeração.
         /*! Detalhes */
         enumVar;


//! Uma função membro para teste.
/*!
 *  \param a um inteiro que representa o tamanho do array.
 *  \param s um ponteiro constante para um array de char.
 *  \return um inteiro.
 *  \sa Test(), ~Test(), testMeToo() and publicVar()
     */
int testMe(int a, const char *s);

};
```