"Doxygeneador C++"

Pablo Jesús Jiménez Ortiz
Elena Cantero Molina

1. Introducción

En ocasiones se escribe código para uno mismo y tras un tiempo puede que se quiera rescatar y darle un uso formal a lo escrito pero... ¿Qué se hace si el código no está propiamente documentado?

Es por ello que es muy agradable tener una utilidad que sea capaz de encontrar cada función de nuestro enrevesado código y generar, para cada una, una estructura de código de documentación del tipo Doxygen.

2. Funcionamiento

El funcionamiento de nuestra máquina es bastante simple. La implementación busca en un fichero que le pasemos (en nuestro caso será siempre un archivo ".cpp") estructuras que correspondan con la de una declaración de una función en C++. Esto es:

$$[tipo\ devuelto] \quad [nombre\ funcion] \quad (\ [parametros]\ )$$

Hay que tener en cuenta que los parámetros en una función pueden existir o no , y que puede haber más de un parámetro. Para encontrar las expresiones que se han dado, los patrones a buscar son:

- Tipo devuelto: Este es el tipo que nuestra función devuelve (los más usuales son void, int, bool...). Sin embargo, también puede que haya tipos que sean creados por nosotros mismos (*clases* o *struct*) o tipos que necesitan de especificación de subtipos (tipos con *templates*). Por tanto, lo que deberá buscar nuestro programa es una palabra compuesta de símbolos cualesquiera que no tenga ningún espacio, salto de línea, tabulador o símbolo de igualdad. Así, podemos capturar cualquier tipo.

- Espacio: Siempre habrá un espacio entre el tipo y el nombre de la función.
- Nombre función: En este caso, será una palabra más simple, pues ya sabemos que los nombres de las funciones y variables no pueden tener caracteres que no sean ni letras ni números (a excepción de la barra baja).
- Espacio y paréntesis: El espacio es opcional, y el paréntesis que indicará que va a empezar el espacio donde se encuentran los posibles parámetros de la función.
- Parámetros: Como ya hemos comentado, pueden ser uno, varios o ninguno. Para ello, nuestro programa deberá comprobar si hay algo antes del siguiente paréntesis y después, mirar si dentro del paréntesis hay comas, que separan los diferentes parámetros.
- Paréntesis de cierre: finaliza la estructura que queremos buscar.

Una vez que nuestro programa encuentre en el fichero las estructuras que cumplen dicho formato, para lo cual utilizamos el lenguaje Lex que nos permite definir las expresiones regulares, escribimos un código en C que nos permite tratar con el patrón encontrado.

Lex define una serie de variables en las que se almacenan datos acerca de la estructura que hemos encontrado de acuerdo a un patrón. El más importante que utilizaremos es la cadena de caracteres "yytext", que almacena en ella la estructura entera. Por tanto, iterando sobre esta cadena de caracteres podremos identificar de forma sencilla cada uno de los componentes de nuestra función, pues sabemos de antemano qué estructura tiene cada uno, y poder así separarlos y conseguir una estructura de la forma:

```
/*
 * @brief Descripcion
 * @return Tipo devuelto
 * @param Parametro 1
 * ...
 * @param Parametro n
 */

tipo_devuelto nombre_funcion (parametro_1, ... , parametro_n)
```

Que es la que tiene normalmente la documentación doxygen.

3. Ejemplo de funcionamiento

Para probar nuestro programa de forma más sencilla, lo que he hecho ha sido que el código "doxygenado"lo introduzca en un fichero aparte. Al hacer esto, no tenemos que estar borrando el texto que el programa añade cada vez que se ejecuta el programa sobre el mismo archivo. El fichero que le pasaremos como entrada a nuestro programa tendrá el siguiente contenido:

```
/*
Ejemplo ilustrativo del uso basico de funciones. Declaracion y
    llamada de funciones tanto con parametros como sin ellos.
*/

#include <stdlib.h>

int funcionuno (){ return 1; }

int funciondos ( int x ){ return x+1; }

int funciontres ( int aa , int ba , int c );

pair<int,char> funcioncuatro ( int a , char ba );

void main ( int arg , char argv ){

    // Genera numero aleatorio entre 1 y 10
    int n = rand()%10 + 1;

    std::vector<int> v;
    // Aniadimos numero generado al final del vector
    v.push_back(n);

    return 0;
}
```

Como sabemos, compilamos el lex primero, nos genera un archivo en c que compilamos y lanzamos el ejecutable. La salida en mi caso será un archivo "funciones.txt."[en] el que el contenido es:

```
/*
 * @brief "Add your own brief"
 * @return int
 */
```

```
int funcionuno ()
/*
 * @brief "Add your own brief"
 * @return int
 * @param int x
 */
int funciondos ( int x )
/*
 * @brief "Add your own brief"
 * @return int
 * @param int aa
 * @param int ba
 * @param int c
 */
int funciontres ( int aa , int ba , int c )
/*
 * @brief "Add your own brief"
 * @return pair<int,char>
 * @param int a
 * @param char ba
 */
pair<int,char> funcioncuatro ( int a , char ba )
/*
 * @brief "Add your own brief"
 * @return void
 * @param int arg
 * @param char argv
 */
void main ( int arg , char argv )
```

Podemos observar cómo selecciona y documenta solamente aquellas funciones que se estén declarando. Finalmente, se obtiene un fichero documentado en formato Doxygen.