Algoritmos y Programación II (75.41) - Cátedra Lic. Gustavo Carolo
1° Cuatrimestre 2009

Trabajo Práctico Nº 1: Abstracción

Condiciones de entrega

El presente trabajo práctico deberá cumplir con todos los requisitos detallados en el reglamento de la cátedra. La fecha de entrega es el martes 6 de abril de 2009.

Documentación automática del código fuente

Se requiere un TDA que lea cada una de las líneas de un código fuente en lenguaje C, de modo que obtenga aquellas que contengan comentarios de documentación, para luego presentarlos a modo de resumen y en un formato determinado sobre un archivo de texto, descartando todas aquellas líneas que no sean de tipo comentario, que se encontraban contenidas en el archivo fuente original como parte de un comentario standard. El formato del documento de salida a generar debe ser texto plano (obligatorio).
De esta forma se podrá obtener la documentación del código fuente automáticamente al procesar el mismo.

Reglas para la documentación:

1. Cada comentario comienza con la cadena de caracteres "/**", y finaliza con la cadena de caracteres "*/". Notar que como los comentarios en lenguaje C comienzan con "/*" y terminan con "*/", los comentarios a utilizar para la documentación automática (AUTO) son un subconjunto dentro de los comentarios tradicionales C, y por lo tanto no serán interpretados por un compilador.

Un comentario AUTO tendrá la siguiente estructura:

```
/**
Esto es un comentario AUTO
*/
```

2. La secuencia /** no tiene por qué estar ubicada al comienzo de una línea (puede haber espacios a la izquierda). De manera similar, una secuencia *****/ no tiene por qué estar ubicada al comienzo de la línea de texto (por practicidad basta verificar si una línea *contiene* los símbolos /* y */ para apertura y cierre de un comentario AUTO, respectivamente).

3. No se permite tener anidamientos de comentarios AUTO, por lo que el siguiente contenido será erróneo:

```
/** Esto es el primer comentario
/** y este es el segundo comentario */
*/
```

4. Si en una línea se abre un AUTO dos o más veces, sólo se considerará la primer apertura.

Ejemplo:

```
/** /** /** comentario */
```

5. Si en una línea se cierra un AUTO dos o más veces, sólo se considerará el primer cierre.

```
Ejemplo: /** comentario */ */ */
```

6. No se conoce de antemano la longitud de una línea de texto.
7. No se conoce de antemano la cantidad de líneas de texto que conforman un AUTO.
8. La primera línea de un AUTO siempre contiene únicamente el tag de apertura de comentario (/**), y ninguna otra información adicional que deba ser considerada, si es que ésta está contenida en la misma línea del tag.
9. La última línea de un AUTO siempre contiene únicamente el tag de cierre de comentario (*/), y ninguna otra información que deba ser considerada, si ésta está contenida en la misma línea del tag.
10. Cada AUTO contiene, después del tag de apertura, una secuencia de parámetros que permite
11. identificar determinadas partes de los comentarios. Estos parámetros estarán bien formados y comienza con el símbolo @.
12. El significado de cada parámetro puede ser consultado en el Diccionario que figura más adelante.
13. A continuación del tag de apertura de un AUTO, en la segunda línea se encuentra un parámetro que identifica a cada uno de los tres tipos de comentarios que pueden presentarse. A saber:

```
/**
@title
...
*/
```

Esto identifica a un comentario AUTO que describe a un título.
Por otra parte, el AUTO

```
/**
@function
...
*/
```

identifica a un comentario referido a una función.
El AUTO:

```
/**
@footer
...
*/
```

identifica a un comentario AUTO que será colocado en el documento final como pie de página.

14. Toda información contenida en la misma línea de @title, @function o @footer, después de estos símbolos, será descartada.

Ejemplo:

```
/**
@function Esto ha de ser descartado
...
*/
```

15. El procesamiento de los AUTOs es case-sensitive, por lo que @footer es distinto de @FOOTER y de @Footer (sólo es válido el primer caso).

16. Toda información concerniente a un parámetro estará contenida en la misma línea, sin importar su longitud. Ejemplo:

```
/**
@function
@name main
@return int
@param void
@description esta función es la principal del programa.
*/
```

17. No habrá renglones en blanco dentro de un AUTO.
18. Dentro de una línea de parámetros AUTO, no está permitido el uso de las cadenas /** y */.
19. Cada comentario AUTO de tipo Title puede contener los siguientes elementos:

@title
@name
@description

xxi) Cada comentario AUTO de tipo Function puede contener los siguientes elementos:

@function
@name
@description
@return
@param
@author
@see

xxii) Cada comentario AUTO de tipo Footer puede contener los siguientes elementos:

@footer
@name
@author
@copyright

Estos tags deberán levantarse de un archivo de configuración con el formato:

```
[tipo_tag]
{
        [elemento1]
        [elemento2]
        …
        [elementoN]
}
```

Obsérvese que el indentado es opcional, se recomienda pero no es obligatorio.
Para el caso actual, por ejemplo:

```
title{
name
description
}
footer{
        name
        author
copyright
}
```

Dado que esta estructura no tiene un límite en cuanto a cantidades, tanto de tags principales como de subtags, asociados a los primeros, se sugiere usar el TDA VariableArray que se encuentra subido en el grupo de la materia. Esta herramienta permite manejar dinámicamente la cantidad de tags que sean necesarias agregándolas a un "vector" sin tener que ocuparse de la alocación y liberación de memoria que queda como responsabilidad del TDA. Queda a criterio del grupo la forma de utilizarlo, y como resolver la anidación de tags dentro de otros, pero deberá ser consultada con el ayudante a cargo para asegurar un uso adecuado.

20. El primer parámetro de un AUTO debe ser: @title, @function o @footer. Los restantes pueden venir en cualquier orden, para ser luegos seguidos por un fin de comentario AUTO (*/).
21. Nunca un parámetro puede formar parte de un comentario monolínea. Ejemplo:

/** @title */

no está permitido.

22. No se conoce de antemano la cantidad de parámetros (argumentos formales) que una función C puede contener, pero a los fines de este TP puede considerarse por simplicidad que esta cifra nunca superará los 50.
23. Todos los parámetros @param estarán uno a continuación del otro. Ejemplo: para la función de biblioteca strcpy(), el programador que desee documentar su código deberá escribir:

```
/**
@function
@name strcpy
@param t: char *
@param s: char *
@return char *
*/
```

24. Los parámetros (argumentos formales) de una función serán indicados con la siguiente sintaxis:

@param <nombre> : <tipo>

siendo el carácter delimitador ":".

El nombre del parámetro está dado por el <nombre>, y su tipo por <tipo>, sin importar lo complejo que éste sea. Ejemplo:

@param cliente : const struct tcliente *

25. Si una función no recibe parámetros, debe indicarse con el parámetro void. Idem para funciones con número variable de argumentos, lo cual se representa con "..." .

Ejemplo: (notar la ausencia del delimitador ":")

```
/**
@function
@name main
@param void
@return void
*/
```

Dicho de otra forma, si se encuentra el delimitador ":" dentro de un @param, la cadena comprendida entre los símbolos @param y ":" representa el nombre del parámetro. Lo que sigue a continuación del delimitador ":" es su tipo. Si no está presente el delimitador ":", lo que sigue a @param es directamente su tipo (void o ...).

26. A los efectos de este TP, todos los comentarios DOC que no tengan un tag identificador válido (@title, @function o @footer) serán ignorados por la aplicación.

27. A los efectos de este TP, todos los comentarios DOC monolínea serán ignorados por la aplicación.
28. Los mensajes de errores que daban hacerse llegar al usuario, deberán manejarse a través de la salida estándar de errores.

Nota: Por cuestiones de simplicidad se ha restringido el tipo de comentario a tres clases (title, function y footer). Si se logra una buena parametrización de código, será mucho más fácil y prolijo extender en un futuro las funcionalidades de este aplicativo

Ejemplos

TDA DocumentoDeCodigo

Diseño del tipo de dato

El diseño del tipo de dato queda a criterio del grupo, y deberá ser presentado al ayudante desingado en la pre-entrega.

Primitivas

Deben desarrollarse las siguientes primitivas para el TDA, respetando exactamente la interfaz y los requerimientos indicados.

int TDC_Crear (TDC* tdc, char* archPrograma)		
FUNCIÓN	Crea un TDA DocumentoDeCodigo asociado a un archivo de código fuente C para su docmentacion automatica.	
PRE	`Tdc` tiene suficiente memoria reservada para una estructura de tipo `TDC`. `archPrograma` es una ruta de archivo válida.	
POST	Si devuelve `0`, `tdc` abierto. Si hubo error al abrir el archivo devuelve `-1`.	
PARÁMETROS	`Tdc`	.
	`archProgram`	Ruta al archivo a documentar.

int TDC_Cerrar(TDC* tdc)		
FUNCIÓN	Cierra el `tdc`.	
PRE	`Tdc` abierto.	
POST	Si devuelve `0`, `tdc` cerrado. En otro caso devuelve `-1`.	
PARÁMETROS	`Tdc`	El manipulador del archivo a documentar.

int TDC_Documentar (TDC* tdc, char* archDoc)		
FUNCIÓN	Genera un archivo con la documentación..	
PRE	`Tdc` abierto. `archDoc` es una ruta de archivo válida.	
POST	Si devuelve `0`, escribió la documentación automática en archDoc. Si devuelve `-1` hubo error.	
PARÁMETROS	`Tdc`	El manipulador del archivo al cual escribir.
	`archDoc`	Nombre del archivo de documentacion.

Programa de aplicación

Se debe desarrollar un programa que, utilizando el **TDA DocumentoDeCodigo** definido más arriba, proponga las siguientes opciones al usuario:

- Documentar un archivo con código fuente C y guardarlo en un archivo del mismo nombre con extensión .txt.

  En este caso el programa se invocará como:

  ```
  tp1 <archFuente>
  ```

  `<archFuente>` es la ruta al archivo con el código fuente C

- Documentar un archivo con código fuente C y guardarlo en un archivo cuyo nombre se indica..

  En este caso el programa se invocará como:

  ```
  tp1 <archFuente> <archDoc>
  ```

  `<archFuente>` es la ruta al archivo con el código fuente C

  `<archDoc>` es la ruta al archivo donde se debe guardar la documentación.

El programa debe informar al usuario si la ejecución fue realizada con éxito o no. **Los mensajes de error deben ser claros y brindar la mayor cantidad de información posible al usuario.** Esto incluye, pero no se limita a, errores de sintaxis, errores originados en el manejo de archivos, y errores al invocar a las primitivas.

Apéndice

Breve explicación del tipo de dato abstracto Variable Array.

Este TDA se encarga de mantener datos en una tira de memoria continua. Este concepto es el mismo de los arreglos (arrays), de ahí el nombre del tipo. Pero al igual que los vectores, está pensado para alojar un conjunto de elementos del mismo tipo, y maneja todo en memoria, por lo que se crea, se usa y se destruye en la misma ejecución de un programa. Sus primitivas permiten agregar elementos, contarlos, obtenerlos, modificarlos y quitarlos de la estructura, pero sólo permiten ocupar posiciones consecutivas desde la primera, es decir, si tiene 2 elementos, no permite agregar elementos en la posición 10, dado que es posible que dicha posición aún no exista, ya que va reservando memoria según se le va requiriendo.

Es indispensable que se lo cree, ya que es indispensable inicializar todos sus campos. Es también muy importante comprobar los errores que puede devolver, sobretodo en inserciones y eliminaciones, dado el intenso uso de las funciones de alocación y liberación de memoria.

Por último, véase la diferencia entre las dos versiones que se pueden encontrar en el grupo. La V1.1, implementa dos funciones de sort con punteros a función, temas que no se explicaron todavía y no son necesarios para el trabajo práctico planteado, por lo que se debe usar la versión 1.0.