SISTEMAS OPERATIVOS

PRÁCTICA 1: LIBRERÍA EN C

ANA M JURADO CRESPO
am.juradoc@alumnos.urjc.es
SISTEMAS OPERATIVOS - URJC - *6 NOV 2019*

CONTENIDOS

AUTORES

Tanto este documento como todos los archivos .c y .h contenidos en el archivo **practica1.zip** han sido realizados por:

El código está bajo control de versiones en GIT:

https://github.com/nukyma/c_programming_files

Ana María Jurado Crespo

am.juradoc@alumnos.urjc.es

DESCRIPCIÓN DEL CÓDIGO

Disponemos de un archivo de cabecera **libreria.h** proporcionado por el profesor. En este archivo encontramos las definiciones de tipos y declaración de funciones de las que se compone nuestra librería.

Hay tres funciones que tendrán que ser implementadas en el archivo fuente **libreria.c**:

1. **int head(int N)**: Se comporta de la misma forma que *head(1)* leyendo de la entrada estándar, es decir, muestra las N primeras líneas en la salida estándar recibidas por la entrada estándar.
2. **int tail(int N):** Se comporta de la misma forma que *tail(1)* leyendo de la entrada estándar, es decir, muestra las N últimas líneas en la salida estándar recibidas por la entrada estándar
3. **int longlines(int N)**: Muestra las N líneas más largas recibidas de mayor a menor, o todas ellas si hay menos de N líneas, por la entrada estándar de forma ordenada en la salida estándar.

Antes de comenzar a implementar las tres funciones que se nos piden, es necesario conocer más sobre las funciones *head(1)* y *tail(1)*, a si que consultamos la bibliografía disponible. Para más información, por favor consulte la sección BIBLIOGRAFÍA.

HEAD Y TAIL

The ***head*** **command** reads the first few lines of any text given to it as an input and writes them to *standard output* (which, by default, is the display screen).

head's basic syntax is:

```
head [options] [file(s)]
```

The square brackets indicate that the enclosed items are optional. By default, head returns the first ten lines of each file name that is provided to it.

The ***tail*** **command** reads the final few lines of any text given to it as an input and writes them to *standard output* (which, by default, is the monitor screen).

The basic syntax for tail is:

```
tail [options] [filenames]
```

The square brackets indicate that the enclosed items are optional. By default, tail returns the final ten lines of each file name that is provided to it.

MEMORIA DINÁMICA

Uno de los requisitos de esta práctica es que usemos memoria dinámica.

Tanto para la función HEAD como para la función TAIL, vamos a usar un buffer de memoria dinámica, por tanto para almacenar las líneas introducidas por entrada estándar (teclado) usaremos un buffer de memoria dinámica.

Para ellos usamos:

- ***malloc***: asigna la memoria solicitada y le devuelve un puntero.
- **free:** libera la memoria "guardada" por malloc

En nuestro caso hemos decidido que el tamaño del buffer sea de 1024 bytes y que el puntero sea de tipo char.

FUNCIÓN HEAD

int head(int N)

Necesitamos implementar una función que se comporte tal y como lo hace la función HEAD descrita anteriormente.

Es necesario que esta función lea las 10 primeras líneas de la entrada estándar (teclado), las almacene en algún sitio y luego las muestre por la salida estándar (pantalla).

Para leer de la entrada estándar he utilizado la función ***getline()*** en vez de usar ***fgets()*** ya que actualmente se desaconseja su uso. La función ***getline()*** lee una línea completa de una secuencia, hasta el siguiente carácter de nueva línea incluido.

size_t getline(char **lineptr, size_t *n, FILE *stream);

- **char **lineptr**: El primero es un puntero a un bloque asignado con malloc o calloc que asigna memoria para el programa cuando se ejecuta. Este parámetro es de tipo char **; contendrá la línea leída por getline.
- **size_t *n**: El segundo parámetro es un puntero a una variable de tipo size_t; Este parámetro especifica el tamaño en bytes del bloque de memoria señalado por el primer parámetro.
- **FILE *stream**: El tercer parámetro es simplemente la secuencia desde donde leer la línea.

En la práctica esta función aparece en la siguiente línea:

charact=getline(&buffer ,&buf_size, stdin);

He incluído las páginas más relevantes de todas las que he consultado al respecto en la bibliografía.

FUNCIÓN TAIL

int tail(int N)

He tenido varias a aproximaciones a la implementación de esta función. A continuación voy a contar a grandes rasgos las ideas que se me fueron ocurriendo por orden cronológico. También voy a comentar los problemas que encontré y las razones por las que deseché esas ideas.

IDEA 1: ARRAY

La primera aproximación a la implementación de esta función que me vino a la cabeza consistía en usar una array de tamaño fijo en el que el índice iba rotando usando una regla algebraica y el contenido del array iba siendo sobre escrito en caso de que fuera necesario.

Lo que me atrajo de esta implementación es concepto de ahorro de memoria ya que no es necesario guardar todas las líneas introducidas. En caso de introducir más líneas de las que luego se han de mostrar, estas pueden ser sobreescritas.

Los problemas que encontré:

- Un array debe definir su tamaño en tiempo de compilación.
- Si el argumento introducido al llamar a la función es un int grande, estamos reservando memoria que no sabemos si vamos a necesitar.

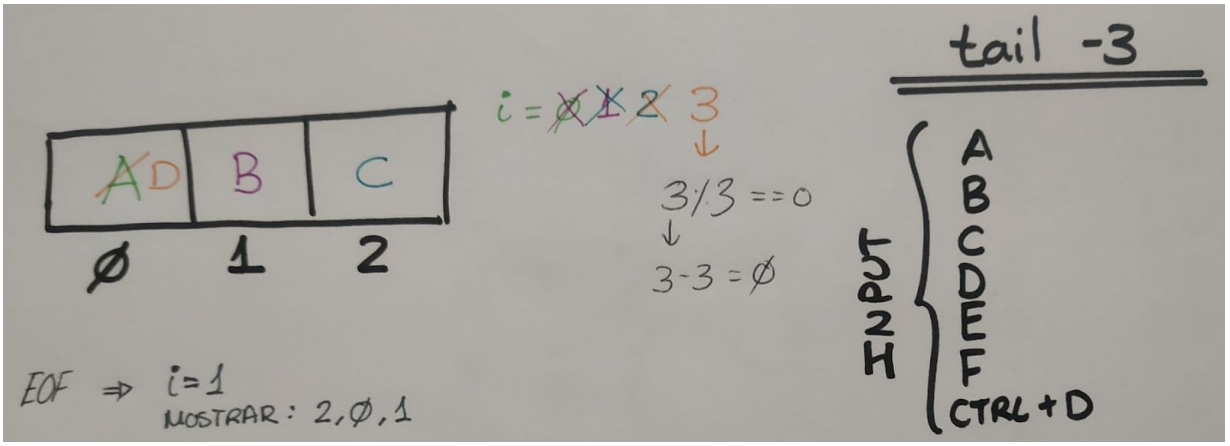


Imagen 1. Dibujo de la aproximación algoritmo.

IDEA 2: MEMORIA DINÁMICA Y MOVER PUNTERO

Teniendo en cuenta los problemas que encontré con la idea del array pensé en que una buena aproximación sería usar memoria dinámica para poder así liberar la memoria tras el uso de la función. Podía recorrer las “posiciones” del array ficticio sumando al buffer.

Haciendo pruebas me dí cuenta de que era posible sumar para conseguir las siguientes posiciones de la memoria reservada con ***malloc***, pero que no funcionaba de la misma manera restando. O al menos a mí no me salía. A si que decidí buscar otra alternativa.

IDEA 3. LISTA DOBLEMENTE ENLAZADA

Basándome en las ideas que tenía, decidí buscar una estructura de datos que me permitiera hacer lo que de alguna forma intuía era la mejor forma de implementar esta función.

En resúmen la función tiene que cumplir lo siguiente:

- Leer de la entrada estándar líneas
- Almacenar esas líneas en algún sitio (buffer circular)
- Al alcanzar N líneas, se irá descartando la primera línea de la cadena, es decir, la linea mas antigua.
- Al alcanzar el fin de la entrada estándar mostrará las líneas últimas N líneas introducidas, es decir las líneas que han quedado almacenadas.

Necesito una estructura de datos que me permita “dar vueltas”.

En un primer momento pensé en una estructura tipo **buffer circular** ó **lista circular**, pero tras pensar un poco en cómo manejar los punteros, decidí seguir buscando.

Finalmente, la estructura de datos que creo que mejor se ajusta a lo que necesito es una **lista doblemente enlazada**.

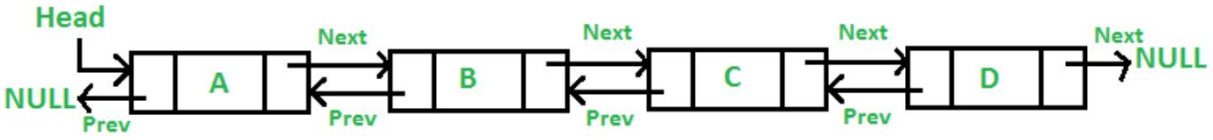


Imagen 2. Representación gráfica de una lista doblemente enlazada

Mirando la siguiente función que hay que implementar **int longlines(int N)** lo que quizás nos venga mejor es una **lista circular doblemente enlazada con inserción ordenada**. Esta es la estructura de datos que creo más adecuada para atacar la implementación de la última función.

Los motivos para usar una estructura circular es que es una ventaja tener localizada la cabecera y la cola de la lista para implementar la inserción ordenada, ya que necesitaremos poder insertar y borrar tanto por la cabecera como por la cola.

Con la inserción ordenada resolvemos el problem de mostrar las x mayores líneas y además mostrarlas de forma ordenada de mayor a menor. QPor extensión, se puede usar este tipo de estructura para resolver a su vez la implementación de **int tail(int N)** usando una función dummy de ordenación.

LISTA CIRCULAR DOBLEMENTE ENLAZADA con INSERCIÓN ORDENADA

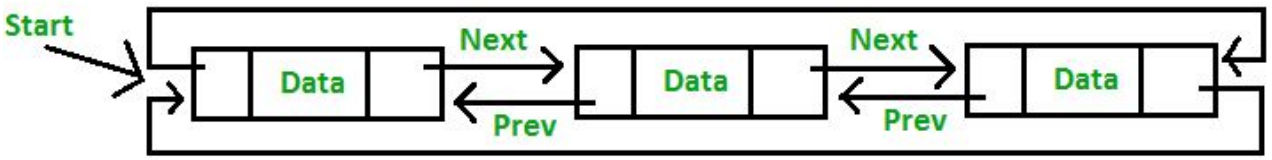


Imagen 3. Representación gráfica de una lista circular doblemente enlazada

Nuevos tipo de datos definidos

Creamos un elemento básico para nuestra lista a la que llamamos **node.**

```
struct node{
	char* line;
	struct node* next_node;
	struct node* prev_node;

};
typedef struct node s_node;
```

Cada nodo consta de tres elementos:

- Un puntero char a ***line***
- Un puntero ***node*** a otro ***node*** (el siguiente de la lista)
- Un puntero ***node*** a otro ***node*** (el anterior de la lista)

La lista doble enlazada con cabecera y cola está compuesta por estos nodos. Mejor dicho por un par de punteros a la cabecera y final de la lista y por el siguiente tipo:

```
typedef int (*d_output_order_function)(char*,char*);
```

Es un puntero a una función (la función para ordenar en nuestro caso) que devuelve un int y tiene como entrada dos char

```
// Tipo de dato nuevo. Lista doblemente enlazada (dllist) con función de ordenación y cabecera (start) y cola (end)
typedef struct doubly_linked_list_start_end_ordered{
	d_output_order_function order_function;
	s_node* start;
	s_node* end;
} s_dllist_start_end;
```

La estructura completa quedaría algo así:

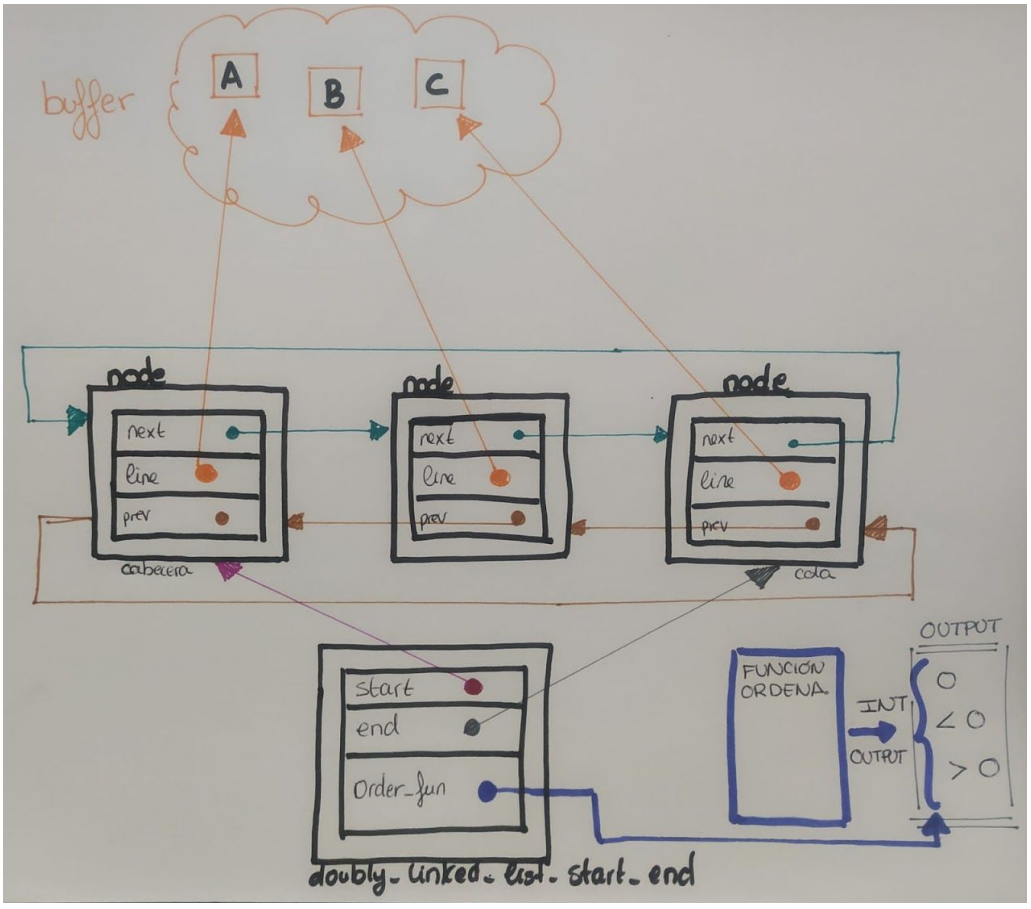


Imagen 4. Estructura de datos customizada

Tenemos una serie de nodos que se conectan a su predecesor y antecesor formando una lista circular y cuyo “contenido” es un puntero tipo char a las líneas que se recogerán de la entrada estándar.

La lista doblemente enlazada con cabecera y cola e inserción ordenada no es más que otro tipo de “nodo” donde guardamos un puntero al nodo cabecera, otro puntero al nodo cola y como “contenido” tenemos un puntero de tipo int que apunta a la salida de una función.

Esta última parte me ha parecido muy interesante ya que no sabía que se podía poner un puntero apuntando a la salida de una función, no a la función en sí misma.

Funciones

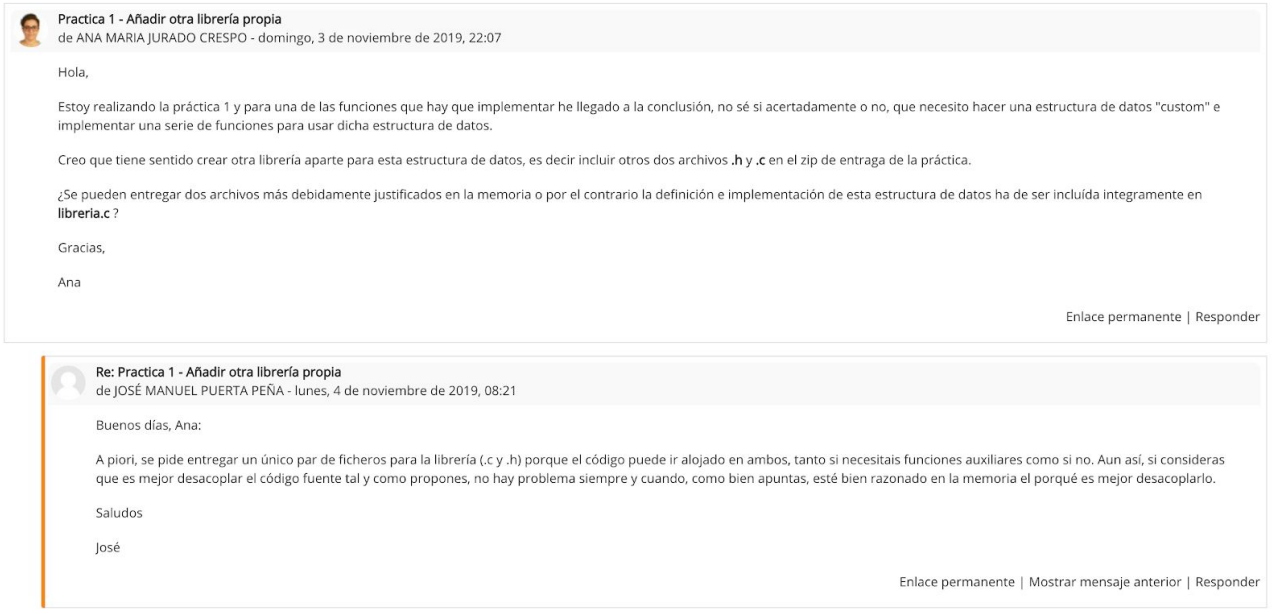
Los métodos que necesitamos para usar esta estructura en nuestras funciones **tail** y **longlines** son:

- Inicializar la lista
  - **s_dllist_start_end* init_list(d_output_order_function);**
- Añadir un nuevo nodo de forma ordenada
  - **void add_node_ordered(s_dllist_start_end* lista,char* new_line);**
- Añadir un nodo al final de la lista
  - **void add_node_end(s_dllist_start_end* lista,char* new_line);**
- Añadir un nodo tras otro nodo en concreto
  - **s_node* add_node_after(s_node* previous_node,s_node* new_node);**
- Borrar el último nodo de la lista (cola)
  - **void delete_node_end(s_dllist_start_end* lista);**
- Borrar el primer nodo de la lista (cabecera)
  - **void delete_node_start(s_dllist_start_end* lista);**
- Borrar la lista entera
  - **void delete_list(s_dllist_start_end* lista);**

Una vez he tenido claro la estructura que quería implementar, la implementación de las funciones no me ha resultado del todo complicada. He seguido las instrucciones de implementación de una lista doblemente enlazada que encontré en GeeksForGeeks. Están añadidas a la bibliografía.

He decidido que el código de esta parte y sus funciones es lo suficientemente complejos, extenso y diferente a lo que se nos pide implementar en libreria.c que lo **he sacado a una librería aparte**. La librería se llama: **lista_doble_enlazada**. Creo que a la hora de corregir el código, tenerlo separado hace que todo sea mucho más legible y ayuda a su comprensión.

Sé que en las especificaciones de la práctica pone expresamente que no se entregen más ficheros que los que se indican, pero pregunté en el foro de la asignatura y el profesor asociado JOSE MANUEL PUERTA PEÑA me respondió lo siguiente:



Practica 1 - Añadir otra librería propia
de ANA MARIA JURADO CRESPO - domingo, 3 de noviembre de 2019, 22:07

Hola,

Estoy realizando la práctica 1 y para una de las funciones que hay que implementar he llegado a la conclusión, no sé si acertadamente o no, que necesito hacer una estructura de datos "custom" e implementar una serie de funciones para usar dicha estructura de datos.

Creo que tiene sentido crear otra librería aparte para esta estructura de datos, es decir incluir otros dos archivos **.h** y **.c** en el zip de entraga de la práctica.

¿Se pueden entregar dos archivos más debidamente justificados en la memoria o por el contrario la definición e implementación de esta estructura de datos ha de ser incluída integramente en **libreria.c** ?

Gracias,

Ana

Enlace permanente | Responder

Re: Practica 1 - Añadir otra librería propia
de JOSÉ MANUEL PUERTA PEÑA - lunes, 4 de noviembre de 2019, 08:21

Buenos días, Ana:

A piori, se pide entregar un único par de ficheros para la librería (.c y .h) porque el código puede ir alojado en ambos, tanto si necesitais funciones auxiliares como si no. Aun así, si consideras que es mejor desacoplar el código fuente tal y como propones, no hay problema siempre y cuando, como bien apuntas, esté bien razonado en la memoria el porqué es mejor desacoplarlo.

Saludos

José

Enlace permanente | Mostrar mensaje anterior | Responder

COMENTARIOS PERSONALES

PROBLEMAS ENCONTRADOS

No sé cçomo evitar que salga la D al hacer CTRL+D para para la inserción de líneas por teclado cuando se vuelven a mostrar en la función tail

En la definición de longlines no se dice qué pasa si al superar el límite marcado se siguen introduciendo líneas de igual longitud a las que ya hay. En mi caso he considerado que esas nuevas líneas no debían ser introducidas en la lista de las que se van a mostrar.

Dar con la estructura de datos adecuada para implementar tail y sobretodo longlines me ha llevado mucho tiempo y muchas búsquedas. Ha sido interesante porque he aprendido por el camino algunas cosas, pero es en lo que más he invertido tiempo.

Quizás el nodo "**doubly_linked_list_start_end_order**" se podría haber evitado usando una variable global donde guardar un puntero al nodo cabecera de la lista. Con esto habría sido suficiente, en principio, para localizar la cabecera y la cola.

El resultado de la función de ordenación podría haberse incluído de otra forma.

De hecho, intenté esta aproximación en teoría más sencilla pero tuve mucho problemas a la hora de indicar qué función de ordenación utilizar si la que ha de usar longlines o la función dummy que siempre devuelve el mismo valor y que se usa para la implementación de tail.

Al final, me decanté por la solución quizás algo más compleja pero que he sabido implementar.

CRÍTICAS Y MEJORAS

He seguido una guía de estilo que he encontrado en internet. Me hubiera gustado que se explicara y se dieran pautas claras de la guía de estilo. Lo he echado en falta a la hora de comentar las funciones de la librería por ejemplo.

He usado la siguiente guía para comentar las funciones C Code Style Guidelines. También he seguido las recomendacione de CS50.

Por motivos de trabajo no puedo asistir presencialmente a casi ninguna clase y hecho mucho en falta que se comenten más cosas sobre las clases y las prácticas en el foro.

Todos los comentarios y recomendaciones que se hacen en clase me las pierdo. Si la opción de dispensa académica está aceptada para esta asignatura, los alumnos que la pedimos no deberíamos estar en inferioridad de condiciones.

En un lenguaje como C, echo de menos tener un entorno de desarrollo más completo donde poder debugear y donde poder ver de forma visual qué variables tengo, de qué tipo son, qué valores tienen, si apuntan a algún sitio a dónde y en qué posición de memoria están guardadas.

No sé debugear en C y creo que es algo que se debería enseñar. Y como entorno de desarrollo he usado Sublime, que no es más que un procesador de textos vitaminado. He tenido que poner y quitar demasiados **printf()** para averiguar lo que estaba haciendo.

TIEMPO DEDICADO

Realizar esta práctica me ha llevado un total de 21h aproximadamente repartidas en 5 jornadas de programación y escritura de la memoria.

BIBLIOGRAFÍA

Información sobre los comandos head y tail:

- The head Command: http://linfo.org/head.html
- The tail Command: http://linfo.org/tail.html
- Bash Head and Tail Command: https://linuxhint.com/bash_head_tail_command/

Documentación general sobre C:

- C documentation: https://devdocs.io/c/
- Memoria Dinámica: http://sopa.dis.ulpgc.es/so/cpp/intro_c/introc75.htm
- GETLINE():
  - Strings in C
  - Archlinux forum
  - C For Dummies - getlin
- Buffer circular:
  - Circular buffers in C: https://embeddedartistry.com/blog/2017/4/6/circular-buffers-in-cc
  - Wikipedia: https://en.wikipedia.org/wiki/Circular_buffer
- Lista doblemente enlazada:
  - Wikipedia: https://es.wikipedia.org/wiki/Lista_doblemente_enlazada
  - Doubly Linked List: https://www.geeksforgeeks.org/doubly-linked-list/
  - Geeks For Geeks articles: https://www.geeksforgeeks.org/data-structures/linked-list/doubly-linked-list/
- Lista circular doblemente enlazada:
  - https://www.geeksforgeeks.org/doubly-circular-linked-list-set-1-introduction-and-insertion/
  - https://www.geeksforgeeks.org/doubly-circular-linked-list-set-2-deletion/
- Typedef: https://publications.gbdirect.co.uk//c_book/chapter8/typedef.html

Guías de estilo en C:

- C Code Style Guidelines: https://www.cs.swarthmore.edu/~newhall/unixhelp/c_codestyle.html
- CS50: https://cs50.readthedocs.io/style/c/#comments
- Nasa Style Guide: https://pdfs.semanticscholar.org/4c0f/60983b227236f3a56332079f0f80086f7d00.pdf