Ayudantía 3 - Estructura de Datos y Algoritmos ELO320

Lectura/escritura de archivos y memoria dinámica

Felipe Vera A.

18 de abril del 2013

1. Manejo de archivos

Si bien C por sí solo no posee funciones para leer o escribir archivos, existen funciones en la librería estandar GNU que sí pueden hacer esta tarea. Dichas funciones se encuentran en la librería `stdio.h`.

El procedimiento para leer o escribir un archivo es el siguiente:

- Abrir el archivo, especificando si se desea leer o escribir el archivo.
- Ingresar o leer los datos que se desee.
- Cerrar el archivo.

Cada uno de estos pasos se detalla a continuación.

1.1. Abrir el archivo

Para abrir el archivo se emplea la función `fopen`. Esta función solicita dos argumentos:

- **Nombre del archivo**, en forma de una cadena de caracteres.
- **Modo**. También en forma de una cadena de caracteres. Puede ser uno de los siguientes:
  - `r`: Solo lectura. El cursor se coloca al principio del archivo.
  - `w`: Solo escritura. El archivo se vacía y se coloca el cursor al principio del archivo.
  - `a`: Añadir datos. Es similar al solo escritura, con la diferencia de que el archivo no se vacía y el cursor se coloca al final del archivo.
  - Existen más modos, pero son raramente ocupados.

`fopen` devuelve un puntero de tipo `FILE*` (un *file stream*), el cual es necesario para las funciones de lectura y escritura de archivos. Devuelve `NULL` si no se logró abrir el archivo (el archivo no existe si se abre en modo `r` o se abre desde un CD o tarjeta SD bloqueada en modo `w`)

1.2. Cerrar el archivo

Para cerrar el archivo se debe aplicar la función `fclose` al `FILE*` de un archivo abierto.

Atención Mientras un archivo esté abierto no puede ser modificado por otros programas (como por ejemplo: no se puede borrar, cambiar nombre, ni extraer la memoria externa en la que se encuentra).

1.3. Lectura

Sirve para obtener la información de un archivo. Esta información puede estar dispuesta de dos formas:

- **Texto:** Es fácil de modificar (se necesita sólo un editor de texto) y comprender. Sin embargo ocupan más espacio y pueden ser más complejos de entender para un programa, dependiendo de qué tan rígida es la estructura del archivo. Ejemplos son estos mismos códigos fuente, páginas web (HTML) y tablas de valores CSV.
- **Binario:** Son difíciles de modificar e interpretar, a menos de que se tenga una documentación clara de qué tipo de contenido tiene el archivo, sin embargo son más fáciles de comprender para un programa. En un binario la información que contiene no es necesariamente texto coherente, sino que puede contener números y otros tipos de datos (inclusive texto) en formato binario (un caracter puede tomar 256 distintos valores, en lugar de 10 si se almacena un número como texto). Además, para ahorrar espacio, muchas veces los contenidos de un archivo binario están comprimidos, lo que hace que un tipo de archivo binario sin documentación sea un verdadero reto de descifrar.

1.3.1. Leer un archivo de texto

Para leer un archivo de texto se puede emplear las siguientes funciones:

- `fscanf (stream, formato, ...)`: Funciona de manera similar a `scanf`, pero acepta un *stream* como entrada. *Si se ocupa un %s como un parámetro a leer, lee hasta el próximo "caracter en blanco" (retorno de línea, espacio, tabulación).*
- `fgets (string, cantidad, stream)`: Una manera para leer el contenido de una línea de archivo y almacenarla en una cadena. Para asegurarse de almacenar el contenido de la línea entera si la capacidad del string es demasiado pequeña se puede usar `getline`.

1.3.2. Leer un archivo binario

`fread (puntero, tamaño_de_cada_objeto_en_bytes, cantidad_de_objetos, stream)`: Almacena los siguientes bytes leidos en el tipo de dato apuntado. Puede ser cualquier tipo de dato, incluso un `struct`. Para proporcionar el argumento `tamaño_de_cada_objeto_en_bytes` se puede usar la macro `sizeof`, la cual devuelve el tamaño de un tipo de dato en bytes.

1.4. Escritura

Sirven para grabar datos en un archivo para su uso posterior.

1.4.1. Escribir un archivo de texto

Existen las siguientes funciones para escribir en un archivo de texto:

- `fprintf (stream, formato, ...))`: Contraparte de `fscanf`, agregando un *stream* como entrada.
- `fputs (string, stream)`: Escribe el contenido de un string en el archivo correspondiente a *stream*.

1.4.2. Escribir un archivo binario

`fwrite (puntero, tamaño_de_cada_objeto_en_bytes, cantidad_de_objetos, stream)`: Es la contraparte de `fread`, orientada a guardar los contenidos de un arreglo apuntado por `puntero` en un archivo.

1.5. Otras funciones

- `feof (stream)`: Indica si el cursor está o no al final del archivo. Es muy útil en rutinas de "lee el archivo hasta el final". Devuelve 0 si el cursor de lectura/escritura no está al final del archivo, y otro número si está al final.
- `fseek (stream, posición, tipo_de_seek)`: Posiciona el cursor de lectura/escritura de acuerdo a los parámetros dados y a la actual posición del cursor. `tipo_de_seek` puede ser uno de las siguientes constantes:
  - `SEEK_SET`: Coloca el cursor en la posición dada en el archivo.
  - `SEEK_CUR`: Avanza el cursor según el parámetro dado.
  - `SEEK_END`: Coloca el cursor al final del archivo.
- `ftell(stream)`: Devuelve la posición del cursor.
- `rewind(stream)`: Coloca el cursor al principio del archivo. Equivalente a `fseek(stream, 0, SEEK_SET)`.

2. Memoria dinámica

Se sabe que en un sistema operativo moderno toda la memoria RAM se organiza en tres porciones destinadas a distintos usos:

- **Memoria estática**: Es el lugar en el que se guardan las variables globales del programa. Se libera al momento de que el programa deja de ejecutarse.
- **Stack**: Se almacenan todas las variables que necesite una función. Se liberan al momento de salir de esa función. [1]
- **Heap**: Es una porción de memoria mucho más grande y representa un pozo de memoria que gestiona el sistema operativo y puede ser asignado a cualquier programa que lo necesite.

La memoria dinámica se basa, precisamente, en trabajar con el *heap*, para lo cual hay distintas funciones en la librería `stdlib.h` que permiten pedirle al sistema operativo asignar un poco de memoria al programa para que la maneje como quiera. Lo bueno de asignar memoria de esta forma es que el programa puede evaluar cuánta memoria necesita reservar y pedir esa cantidad al sistema operativo. Ni más ni menos.

2.1. Reservar memoria dinámica (malloc y free)

Se hace mediante la función `malloc (cantidad_de_memoria)`. A la función se le pasa como argumento la cantidad de memoria a reservar (recuerda la macro `sizeof`), tras lo cual devuelve un puntero que apunta a dicha memoria, la cual puede ser trabajada como un arreglo.

¡Importante! Luego de haber terminado de ocupar la memoria reservada con `malloc`, se debe solicitar al sistema operativo que la libere usando la función `free (puntero_malloc)`. De lo contrario esta memoria no podrá ser ocupada por otros programas en ejecución, perjudicando su rendimiento.

Nota Reservar memoria con `malloc` no garantiza que su contenido sean ceros. Para hacer eso existe una función llamada `calloc` que asegura que la memoria reservada se llena con ceros.

[1] Se les invita a investigar también la función `alloca`, que en ciertas ocasiones puede ser más conveniente que `malloc`.

2.2. Listas enlazadas

Es una forma de crear un "arreglo de tamaño variable". Es útil si se necesita manejar una cantidad desconocida de elementos.

Se puede acceder a la lista enlazada mediante un puntero. Cada elemento estará encadenado mediante punteros al siguiente hasta que uno de ellos sea un puntero nulo.

En C se implementan como estructuras.

```
struct lista
{
    int dato; //Puede ser de cualquier tipo.
    struct lista *next;
};

struct lista *milista = NULL;    //Una lista vacía
```

Cuidado Si no se especifica explícitamente que un puntero se inicialice con `NULL` puede tomar cualquier valor, lo que puede generar inconvenientes en rutinas que recorran la lista.

2.2.1. Crear un elemento

Para crear un elemento se debe reservar la memoria correspondiente a ese elemento usando `malloc`. La forma de insertar este elemento al primer lugar de la lista es:

- Reservar memoria necesaria para el nuevo elemento (`malloc (sizeof (struct lista))`)
- Si se quiere insertar una cadena o un arreglo dentro del elemento, reservar la memoria necesaria para estas.
- Guardar el puntero al *anterior* primer elemento en una variable temporal.
- Hacer que el puntero hacia el primer elemento apunte al elemento recién creado.
- Hacer que el puntero `next` del elemento recién creado tenga el mismo valor de la variable temporal.

2.2.2. Liberar la memoria ocupada por la lista

Para ello se necesita liberar la memoria ocupada por cada elemento. Se debe iterar el siguiente procedimiento.

- Se verifica que el elemento no sea `NULL`. Si lo es, se termina la rutina.
- Guardar el puntero `next` en una variable temporal.
- Si se asignó memoria (`malloc`) a otros arreglos dentro de la lista, liberarlos usando `free`.
- Liberar la memoria del elemento usando `free`.
- Repetir el procedimiento con el elemento apuntado por la variable temporal.

Ejercicios

1. ARCHIVOS DE TEXTO: LECTURA Se incluye dentro del comprimido un archivo llamado *planilla.csv*. Lea el archivo hasta el final y promedie los datos de cada columna. Imprima los resultados.

2. ARCHIVOS BINARIOS: LECTURA Se incluye dentro del comprimido un archivo llamado *imagen.bmp*. Se puede encontrar una descripción de la cabecera de un archivo BMP aquí: . Cree una función que lea la cabecera, determine si el archivo es, efectivamente, una imagen BMP e imprima los datos de la cabecera en un archivo llamado *resultados.txt*.

3. LISTAS ENLAZADAS En el archivo adjunto *catalogo.txt* se incluye el catálogo de precios y productos de un supermercado, separado por tabuladores. Cree un programa que solicite los números de cada producto para incluirlos en un carrito de compra; y cuando se ingrese un cero, se debe arrojar el total de la compra y los productos que fueron seleccionados. Ese resultado debe ser impreso en un archivo llamado *compra.txt* (Sugiero emplear *structs* para leer el archivo). Si le queda tiempo, intente modificar el archivo *catalogo.txt* para que incluya más productos.