Programación Orientada a Datos - ANSI C

Organización del Computador II

Segundo Cuatrimestre 2023

En este taller vamos a trabajar con código `C` interpretándolo desde la perspectiva de los datos, y en particular, de la forma en que los datos se ubican en memoria. Implementaremos algunas estructuras de datos que nos ayudarán a entender cómo programar en el lenguaje C y cómo hacer correcto uso de memoria dinámica.

Al corregir cada **checkpoint** todos los miembros del grupo deben estar presentes, salvo aquellas instancias en que puedan justificar su ausencia previamente. Si al finalizar la práctica de la materia algún miembro cuenta con más de un 20 % de ausencia no justificada en la totalidad de los checkpoints se considerará la cursada como desaprobada.

Cada checkpoint se presenta a lx docente asignadx durante el transcurso de la clase práctica actual o la siguiente, al igual que en la primera actividad, se recomienda subir los archivos completos al repositorio git.

Introducción

En el campus encontrarán el archivo `ejC2-bundle.v0.1.tar.gz` conteniendo el presente enunciado junto con el código fuente (incompleto) de varias funciones que deberán implementar.

El **primer ejercicio** es una breve introducción al manejo de strings (cadenas de caracteres) en `C`. El **segundo ejercicio** consiste en implementar dos estructuras de datos diferentes. La primera es una **lista enlazada** donde cada nodo almacena un array de enteros y su respectiva longitud. La segunda es un **vector de enteros**, con la particularidad de que sea dinámico, es decir le podemos "pushear" elementos y la estructura misma se encargará de redimensionar su propio tamaño. Por último, con el **tercer ejercicio** se busca integrar los temas vistos hasta ahora, donde deberán trabajar con **arrays de strings**.

Para cada uno de los ejercicios se cuenta con tests para que pueden ejecutar y así asegurarse de que sus implementaciones funcionan correctamente. Para aquellos ejercicios que requieran el uso de memoria dinámica, los tests además de chequear el correcto funcionamiento del programa también se asegurarán de que este haga uso adecuado de la memoria, es decir no esté perdiendo memoria ni accediendo a posiciones que no debería.

1. Precalentando

En este breve checkpoint primero vamos a introducirnos en el manejo de strings en `C`. Lo importante será que aprendamos cómo funcionan y comprender el uso de la memoria dinámica.

1.1. Contar Espacios

1. En el archivo `contar_espacios.h` se definen las dos funciones que se deberán implementar, tales implementaciones deben hacerse en el archivo `contar_espacios.c`.

2. Una vez implementadas las funciones pueden correr los tests correspondientes, para eso en la Terminal deberán ejecutar `make tests_contar_espacios` y luego `./tests_contar_espacios`

Checkpoint 1

2. Complejizando un poco

2.1. Lista Enlazada

En el archivo `lista_enlazada.h` se define la estructura que modela la lista que queremos implementar y se encuentran los prototipos de las funciones que se deberán programar.

1. Realizar un esquema ejemplificando la estructura de la lista.

2. Si dentro de una función cualquiera, creamos una lista haciendo `lista_t* mi_lista = nueva_lista();`, y luego otra haciendo `lista_t mi_otra_lista;` ¿en que segmentos de memoria se alojan las siguientes variables?:

   a) `mi_lista`

b) `mi_otra_lista`

c) `mi_otra_lista.head`

d) `mi_lista->head`

¿Y si a la lista `mi_otra_lista` la creamos fuera de cualquier función?

3. Implementar las funciones definidas en el archivo `lista_enlazada.h`. Tales implementaciones deben hacerse en el archivo `lista_enlazada.c`. Una vez implementadas las funciones pueden correr los tests correspondientes, para eso en la terminal deberán ejecutar `make tests_lista_enlazada` y luego `./tests_lista_enlazada`. Luego, para correr tests con checkeo de memoria ejecutar: `make run_tests_lista_enlazada` (no autocompleta).

2.2. Vector

En `C`, los arrays no son más que un puntero que apunta a una dirección de memoria. El lenguaje no nos otorga ninguna forma de especificar en el mismo tipo de dato su longitud, a diferencia de otros lenguajes donde además de especificar quá datos pertenecerán al array también podemos decirle el tamaño que este tendrá.

Una forma de declarar un array estático, por ejemplo dentro una función sería:

```
void una_funcion(void) {
    ...

    int mi_array[10];

    ...
}
```

Aquí, estamos declarando la variable `mi_array`, la cual sera de tipo `int *`, que apuntará a una dirección de memoria, a partir de la cual tendremos en forma contigua 10 elementos de tipo entero (`int`).

El problema está en que al momento de usar la variable `mi_array`, por ejemplo para acceder a algún elemento del array, en ningún momento se hará un chequeo de que tal acceso sea válido. Simplemente intentará acceder a esa posición de memoria.

En este ejercicio implementaremos una estructura de datos que nos permita reemplazar el uso de arrays por algo más seguro, es decir que verifique que la posición de memoria que se intenta acceder sea válida.

Además, nos gustaría que tal estructura de datos sea dinámica, en el sentido que le podamos ir agregando elementos sin saber de antemano cuánto espacio necesitaremos. Dicha estructura se la conoce comúnmente como Vector.

1. En el archivo `vector.h` se define la estructura correspondiente al vector que estamos implementando, y adicionalmente las funciones que se deberán programar. Tales implementaciones deben hacerse en el archivo `vector.c`.
   Una vez implementadas las funciones pueden correr los tests correspondientes, para eso en la Terminal deberán ejecutar `make tests_vector` y luego `./tests_vector`. Luego, para correr tests con checkeo de memoria ejecutar: `make run_tests_lista_enlazada` (no autocompleta).

Checkpoint 2

3. Ejercicio Integrador

La idea de este ejercicio es integrar todos los temas vistos hasta ahora. La correcta implementación implica un entendimiento del manejo de strings como de memoria dinámica en el lenguaje `C`.

El ejercicio consiste en lo siguiente:

Tenemos varios strings y queremos clasificar sus caracteres, es decir analizar cuáles de ellos son vocales y cuáles consonantes. Una vez hecho este análisis, nos interesará devolver estos caracteres ya clasificados en dos arrays distintos, uno para las vocales y otro para las consonantes.

La función que se debe implementar tiene la siguiente aridad:

```
void classify_chars(classifier_t* array, uint64_t size_of_array);
```

El primer parámetro es un array de `classifier_t`, y el segundo la longitud de dicho array. `classifier_t` es una estructura con la siguiente forma:

```
classifier_t {
    char** vowels_and_consonants;
    char* string;
};
```

Entonces, a la función `classify_chars` se la llamará con un array de `classifier_t`, que'**sólo** tendrán seteado su campo `string`, el campo `vowels_and_consonants` estará apuntando a `NULL`.

La función deberá configurar el campo `vowels_and_consonants` como corresponde. Esto es:

- Que en el primer array (indicado por `vowels_and_consonants[0]`) queden todas las vocales del string.
- Que en el segundo (`vowels_and_consonants[1]`) queden las consonantes.

Tener en cuenta que el puntero `vowels_and_consonants` nos viene apuntando a `NULL`, por lo tanto, antes de configurar los arrays, habrá que pedir la memoria correspondiente.
Se puede asumir que cada string nunca tendrá más de 64 vocales ni más de 64 consonantes. Es necesario inicializar la memoria pedida con `0` (cero) de manera que el final del string quede correctamente delimitado. *Hint:* investigar `memset` y `calloc`.
Una vez implementadas las funciones pueden correr los tests correspondientes, para eso en la Terminal deberán ejecutar `make tests_classify_chars` y luego `tests_classify_chars`. Finalmente, para correr tests con checkeo de memoria ejecutar: `make run_tests_classify_chars` (no autocompleta).

Para cerrar:

1. ¿Por qué cuándo declaramos un string en C no hace falta especificar su tamaño de antemano?
2. Supongamos que ahora nos interesa implementar una función para retornar ambas listas de vocales y consonantes. El lenguaje `C` no nos provee ninguna forma sintáctica de retornar más de un dato de una función. Explorar distintas formas que se podría resolver este problema (al menos dos formas distintas, y que funcionen).

Checkpoint 3