FACULTAD DE
INGENIERÍA Y CIENCIAS

UNIVERSIDAD ADOLFO IBÁÑEZ

Tarea 1

Lenguaje y Paradigmas de Programación

Alumnos: Ignacio León
Alonso Tamayo
Filomena Tejeda

Profesor: Justo Vargas
Fecha: 5 de abril del 2025

1 Introducción

Esta tarea tiene como objetivo aplicar los conocimientos adquiridos en clases de Lenguaje y Paradigmas utilizando el lenguaje **C**. Se plantea el desafío de desarrollar un programa capaz de leer un archivo CSV que contiene registros de ventas de una pizzería y calcular distintas métricas básicas a partir de esa información.

El programa debe ejecutarse desde la consola, recibiendo como argumentos el nombre del archivo y las métricas que se desean calcular.

```
./app1 ventas.csv pms pls dms dlsp apo ims ...
```

Entre las métricas solicitadas se encuentran, por ejemplo, la pizza más vendida, la fecha con mayores ingresos, o el ingrediente más popular. Cada métrica fue implementada como una función independiente y ejecutada de manera dinámica utilizando punteros a funciones, lo que permite un diseño más flexible y ordenado.

Una de las principales exigencias del proyecto es organizar el código en varios archivos fuente, de forma modular y lógica. Esto busca no solo facilitar la lectura y el mantenimiento del programa, sino también promover buenas prácticas de desarrollo, como la separación de responsabilidades y el uso adecuado de cabeceras. Además, se pide trabajar utilizando herramientas complementarias como **GIT** para el control de versiones y **Makefile** para la automatización de la compilación.

El enunciado establece ciertos supuestos que simplifican la implementación, como asumir que los datos de entrada son válidos y que el archivo CSV no contiene campos vacíos. Gracias a esto, el enfoque principal está en la correcta lectura de los datos, el procesamiento eficiente de la información y la implementación precisa de las métricas solicitadas.

Más allá de cumplir con los requerimientos técnicos, esta tarea busca reforzar habilidades fundamentales como el diseño limpio de programas, el uso eficiente de la memoria dinámica y la preparación para proyectos más complejos. También permite acercarse a una forma de trabajo más profesional, en la que la organización del proyecto y la claridad del código son tan importantes como la funcionalidad misma.

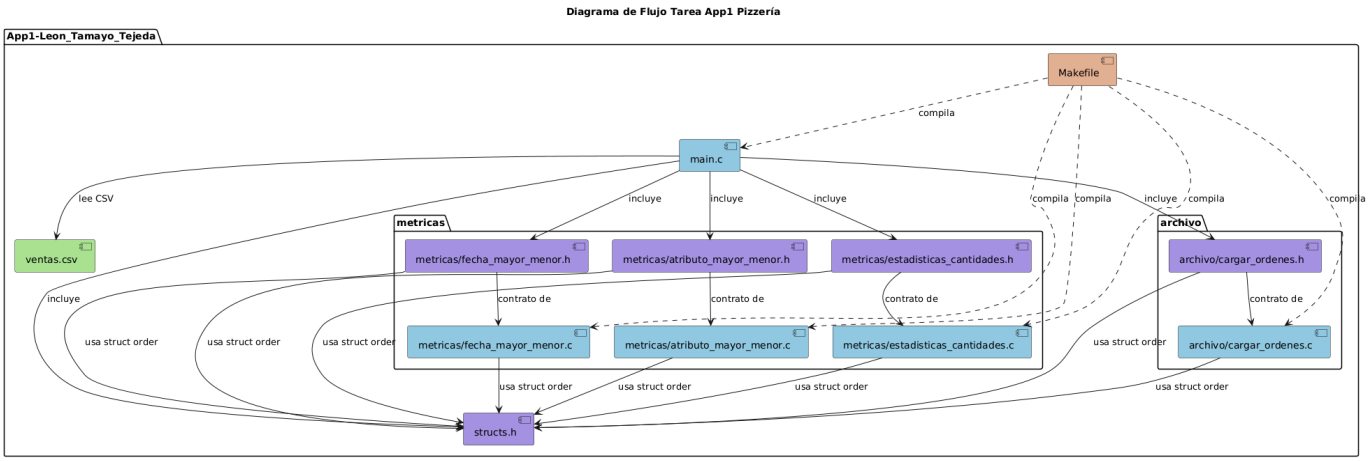
2 Organización del código

El proyecto se organizó en varias carpetas para que cada parte tuviera un propósito específico y bien definido. En primer lugar, en la carpeta principal se encuentran tres archivos clave. "`main.c`" es el archivo principal que maneja el flujo del programa, gestionando las operaciones con los datos de entrada, que contienen la estructura de las órdenes. Además, es el encargado de gestionar los punteros a las funciones de las métricas. "`Makefile`" facilita el proceso de compilación, asegurándose de que todos los archivos necesarios se compilen correctamente. "structs.h" define la estructura de datos utilizada para almacenar la información de cada venta, permitiendo que el programa trabaje con estos datos de manera eficiente.

Dentro de la carpeta principal, se encuentran dos subcarpetas. La carpeta "`archivo`" contiene los archivos "`cargar_ordenes.c`" y "`cargar_ordenes.h`". Estos archivos son los encargados de leer el archivo CSV que contiene los datos de ventas y convertir esa información en estructuras que el programa pueda procesar.

La otra subcarpeta es "`metricas`", que agrupa seis archivos distribuidos en tres grupos relacionados con las métricas que el programa debe calcular. Cada grupo tiene un archivo de cabecera (**.h**) y un archivo de implementación (**.c**). Los grupos son: "`atributo_mayor_menor`", que incluye funciones para calcular métricas como la pizza más vendida, la pizza menos vendida y el ingrediente más vendido; "`estadisticas_cantidades`", que calcula promedios por día y orden, y cantidades de ventas por categoría; y "`fecha_mayor_menor`", que determina las fechas con más o menos ventas en términos de dinero y cantidad de pizzas.

De esta manera, la organización del código permite que cada archivo y cada carpeta tenga una función clara y específica, facilitando la comprensión y el mantenimiento del programa, además de que se puede extender de manera sencilla si se requiere agregar nuevas funcionalidades en el futuro.



3 Estructuras de datos empleadas

Para representar la configuración de las órdenes, se definió una estructura con **“typedef struct”** llamada **“order”**, que encapsula los datos de cada fila del archivo CSV. Esta estructura contiene información como el ID de la pizza, la cantidad, la fecha de la orden, el nombre de la pizza, la categoría, entre otros, con sus respectivos tipos de variables.

Además, se utilizaron estructuras auxiliares para facilitar la gestión de los datos. Por ejemplo, se definió una estructura **“typedef struct”** llamada **“Categoria”** que se encuentra en el archivo de “`estadisticas_cantidades.c`”, que lleva el conteo de las pizzas por categoría. También se creó la estructura de tipo **“typedef struct”** llamada **“VentaFecha”** localizado en el archivo “`fecha_mayor_menor.c`”, que almacena las pizzas vendidas y el dinero generado por fecha, para la fecha con más y menos ventas e ingresos.

Asimismo, se emplearon estructuras de tipo arreglo simple, tanto de enteros como de cadenas de texto, para almacenar, por ejemplo, los nombres de las pizzas o los ingredientes, junto con sus respectivas cantidades. Los arreglos dinámicos de estas estructuras fueron utilizados para almacenar múltiples registros en memoria, lo que permitió su posterior manejo y uso local dentro de sus respectivas funciones del programa.

4 Justificación de la modularidad y el uso de punteros a funciones

Parseo del CSV y almacenamiento de ingredientes:

Se utilizó fopen() y fclose() para abrir en modo lectura y cerrar el archivo CSV. También, se usó fgets() junto con sscanf() para parsear cada línea del archivo, lo que permite un control detallado del formato de entrada y evita dependencias externas, definiendo cada variable separada por una coma de la primera línea con su respectivo tipo de variable, ya sea “`char`”, “`string`”, “`int`” o “`float`”, en particular se utilizaron “`string`” y “`float`”.

Para el caso específico de los ingredientes fueron almacenados como cadenas de texto en el campo pizza_ingredients dentro de order, el que se diferenció por contenerse entre comillas, de esta forma se almacenó como una cadena de texto separada por comas, el que dentro de la función ingrediente_mas_vendido() se realiza la separación de cada tipo de ingrediente, y así poder realizar las operaciones necesarias para obtener el ingrediente más vendido, que en el fondo es la cantidad que se repite en cada orden según cuántas pizzas en total lo contienen.

Modularidad:

Después de estudiar las métricas solicitadas, con el objetivo de no crear un archivo (“módulo”) para cada una, dado que eran bastantes, y que esta estructura de archivos podría hacer más lenta la expansión del código si eso llegaba a ser necesario; se concluyó que lo mejor sería separar las métricas en archivos que agruparan a una acotada selección de estas, según una categoría más grande que las identificara fácil y certeramente, estos archivos se guardaron en una carpeta dedicada a estos con la siguiente distribución y categorías:

- Categoría Atributo Mayor y Menor (archivos atributo_mayor_menor.c y atributo_mayor_menor.h): esta categoría contiene a las funciones asociadas a los parámetros **pms**, **pls** e **ims**, que corresponden a **pizza más vendida**, **pizza menos vendida**, e **ingrediente más vendido**, respectivamente.
  Se eligió este grupo ya que el resultado que debían regresar correspondía a calcular solamente una moda entre los datos de una columna particular (o el opuesto de una moda en el caso de pls), lo que posteriormente ayudó a nombrar a la categoría en el proyecto como Atributo Mayor y Menor.
- Categoría Fecha de Mayor y Menor (archivos fecha_mayor_menor.c y fecha_mayor_menor.h): esta categoría contiene a las funciones asociadas a los parámetros **dms**, **dls**, **dmsp** y **dlsp**, que corresponden a **fecha con más ventas en términos de dinero**, **fecha con menos ventas en términos de dinero**, **fecha con más ventas en términos de cantidad de pizzas**, y **fecha con menos ventas en términos de cantidad de pizzas**, respectivamente.
  Se eligió este grupo rápidamente por la palabra que antecede a las cuatro métricas: fecha, por lo que este grupo necesitaría funcionalidades similares en cuanto a la interpretación de fechas, además de que el conteo y resultado final dependía de una fecha (la que debía ser mostrada también).
- Categoría Estadísticas y Cantidades (archivos estadisticas_cantidades.c y estadisticas_cantidades.h): esta categoría contiene a las funciones asociadas a los parámetros **apo**, **adp** y **hp**, que corresponden a **promedio de pizzas por orden**, **promedio de pizzas por día**, y **cantidad de pizzas por categoría vendidas**, respectivamente.
  Se eligió este grupo ya que las tres métricas que luego conformaron este grupo parecían hacer alusión a una característica estadística de los datos, como el promedio según una columna, y la cantidad de pizzas por categoría vendidas parecía aludir a ser utilizada como una visualización de los datos de manera generalizada y fácil de observar, por lo que esta categoría se pensó como el grupo de métricas más importante al momento de hacer un análisis estadístico y con mayor aporte de valor al momento de tomar decisiones accionables, por encima de las otras métricas.

Además de los archivos relacionados a las categorías de las métricas, la modularidad del programa también alcanzó a la lectura de datos, por lo que se creó una carpeta dedicada a los archivos, que contuvo a los archivos cargar_ordenes.c y cargar_ordenes.h. La función de estos archivos es autoexplicativa: carga las órdenes en el puntero preparado que se le entrega como argumento con los datos que lee del archivo CSV que recibe en otro argumento. La decisión de dedicar archivos enteramente a esta parte del programa fue para dar el espacio para poder añadir otras formas de cargar las órdenes en el puntero que entrega main.c en el programa principal, como puede ser en un futuro una lectura desde un archivo excel, una posible base de datos, etc., por lo que esta decisión fue impulsada por escalabilidad, mantenibilidad, y fácil uso del programa.

El uso de punteros a funciones ocurrió únicamente en el archivo main.c, específicamente en la gestión de las funciones de las métricas, y para facilitar su ejecución asociándolas a las cadenas de textos esperadas como parámetros. La forma en la que se implementó esto es la siguiente: primero se crearon dos cosas: un tipo de dato que sería el puntero a una función de una métrica, y una estructura de datos que guardaría lo necesario de una, que en este caso fue el puntero a la función asociada, la

lista de strings que se aceptarían como parámetro en la línea de comando para asociar a esa función de métrica, y la cantidad de estos strings aceptados. Luego de tener estos dos tipos de datos, se creó una lista de estas estructuras de métricas, cada una correspondiendo a una métrica solicitada para el programa, por lo que la lista contenía 10 estructuras de métricas, cada una con el puntero a la función asociada, la lista de strings que se aceptarían como parámetro para mostrar dicha métrica, y la longitud de esta lista de parámetros reconocidos. Ya con estas estructuras de métricas listas, se realiza en el programa principal un ciclo triple, primero: se itera por cada argumento entregado en la línea de comando después del nombre del programa y del nombre del archivo, para sólo iterar sobre los parámetros de las métricas, luego, por cada uno de estos parámetros, se itera sobre cada estructura de métrica, para finalmente ingresar al tercer ciclo que itera por cada cadena de texto en la estructura de métrica actual que se asocia a esta, y se compara con el parámetro ingresado en la línea de comando actual, para así, si hay coincidencia, se llama a la función asociada a la métrica usando el puntero a la función asociada guardada, con los parámetros estandarizados por la firma de estas funciones, y finalmente se imprime la cadena de texto resultante.

La interacción entre los archivos se basa en crear un header, o sea un archivo "`.h`" que define claramente las estructuras de las funciones a implementar dentro del archivo "`.c`", por lo que el header es como un contrato que manda el formato y maneja los tipos de datos que se usarán dentro de las funciones. En concreto, "`strucs.h`" define la estructura de order que se incluye en cada archivo "`.c`" que utilice estos datos, y particularmente cada categoría de funciones incluye su respectivo header para su correcta inclusión en el programa principal. Esta separación se realiza para tener un mejor manejo de los errores y mantenimientos del programa, de esta forma es sencillo aislar un problema para resolverlo.

5 Herramientas utilizadas:

Para la realización de este código, se utilizaron varias herramientas y páginas web especializadas que facilitaron la implementación de diferentes funcionalidades

Se recurrió a diversas páginas web para profundizar en el uso de funciones específicas de las bibliotecas estándar de C, como stdio.h, stdlib.h y string.h. Estas fuentes proporcionaron información detallada sobre las funciones más utilizadas y sus aplicaciones en el contexto del proyecto (IBM, 2024). Además, se investigaron aspectos relacionados con la instalación y estructura de un Makefile, centrándose en su creación, organización y la comprensión de su funcionamiento interno, así como la estructura del código necesaria para su correcta implementación (Sandeep, 2021) (Stallman et al., 2023). Finalmente, se llevó a cabo una investigación sobre los punteros a función y su implementación dentro del programa, lo que permitió comprender su funcionamiento y cómo aprovecharlos de manera efectiva en la tarea (Yuval, 2009).

A lo largo del desarrollo del código, se utilizó ChatGPT como una herramienta para obtener ayuda en diversas áreas del proyecto. En primer lugar, la IA proporcionó sugerencias sobre la estructura, ayudando a organizar mejor las funciones auxiliares y facilitando el manejo de datos localmente, especialmente cuando se requería almacenar nuevos registros o realizar operaciones adicionales sobre ellos. Además, ChatGPT fue fundamental para la solución de errores, particularmente en la parte del código que involucra la lectura y procesamiento del archivo CSV, donde se identificaron y corrigieron problemas de formato y asignación de variables.

La IA también jugó un papel importante en la explicación de errores y en la aclaración de conceptos complejos relacionados con el uso de funciones de C y el manejo de punteros. A medida que surgían dudas sobre el funcionamiento de ciertas partes del código, ChatGPT proporcionó explicaciones claras que permitieron una mejor comprensión de las implicaciones de las decisiones de programación.

En cuanto a la validación de la información, se cruzaron las sugerencias de ChatGPT con la documentación oficial de las bibliotecas de C, tutoriales de fuentes confiables y el conocimiento previo del equipo. Las ideas propuestas por la IA fueron evaluadas en función de su coherencia con los requisitos del proyecto y su integración con el código existente. Las soluciones de ChatGPT se modificaron o descartaron cuando se consideraron inapropiadas o no alineadas con los objetivos del proyecto, asegurando que el código final fuera una combinación de aportes propios y de la herramienta, sin depender exclusivamente de las sugerencias de la IA.

6 Reflexiones finales

Lo más interesante fue lograr estructurar un programa en C que procesa un archivo CSV relativamente complejo sin el uso de bibliotecas externas, utilizando únicamente las herramientas estándar del lenguaje. Esto permitió reforzar habilidades fundamentales de programación en C.

La parte más desafiante fue, sin duda, el parseo de líneas con `sscanf()`, sobre todo cuando se trataba de variables que incluían comillas, comas internas o formatos mixtos como fechas, cadenas de texto con espacios y cantidades numéricas. Esto exigió un manejo muy cuidadoso de formatos, delimitadores y buffers intermedios para evitar errores de lectura o desbordamientos de memoria.

También resultó interesante trabajar con métricas que implican agrupar datos por campos como `order_date` o `pizza_category`. Esto nos forzó a implementar estructuras auxiliares y desarrollar lógica de búsqueda manual, ya que no se contaba con el soporte de estructuras más sofisticadas como diccionarios o mapas hash que existen en otros lenguajes de más alto nivel.

La depuración se basó en un enfoque paso a paso, utilizando impresión de valores clave con `printf()` para inspeccionar el estado de estructuras, contadores y cadenas en diferentes momentos del flujo. Además, se desarrollaron pruebas incrementales: primero se validó la carga del archivo CSV, luego el parseo de campos individuales, y finalmente la precisión de cada métrica.

También fue importante prestar atención al manejo de la memoria durante el desarrollo del programa. Errores como el uso de punteros sin inicializar, accesos fuera de rango o la omisión de liberar memoria después de su uso fueron detectados mediante un análisis cuidadoso del flujo del programa. La detección temprana y la corrección de estos errores resultaron claves para asegurar la estabilidad y eficiencia del software.

La práctica de modularizar el código fue clave para facilitar el proceso de testing. Al separar funciones como `cargar_ordenes()`, `fecha_mas_vendida()` o `ventas_categoria()`, fue posible probarlas de forma individual utilizando conjuntos de datos reducidos. Esto permitió identificar y corregir errores de manera más rápida y ordenada, además de mejorar la claridad y la organización general del proyecto.

Implementar esto en C reforzó varias lecciones fundamentales del paradigma procedural y de la programación en bajo nivel. En primer lugar, **el manejo de memoria importa**: a diferencia de lenguajes de alto nivel, en C es responsabilidad del programador reservar y liberar memoria manualmente. Una omisión en este proceso puede llevar fácilmente a fugas de memoria o errores de segmentación.

En segundo lugar, **el diseño modular es esencial**: dividir el programa en funciones específicas y organizarlas en archivos separados no solo mejora la legibilidad del código, sino que también facilita su reutilización y mantenimiento a largo plazo.

Además, **la validación de la entrada es crítica**: una sola línea mal formateada en el archivo CSV puede provocar fallos graves si no se verifica adecuadamente el retorno de funciones como `sscanf()` o `fgets()`.

Finalmente, **programar en C obliga a pensar en los detalles**. Desde la longitud de los buffers hasta el orden en que se procesan las estructuras, el lenguaje C requiere una atención rigurosa al detalle que refuerza la disciplina como programador.

Aunque la sintaxis de C es relativamente simple, esta **simplicidad no equivale a facilidad**. El poder que ofrece el lenguaje conlleva una gran responsabilidad en cuanto al control del flujo de ejecución, el manejo manual de la memoria y la gestión adecuada de las estructuras de datos.

Referencias bibliográficas

IBM. (07 de octubre de 2024). < stdio.h >. https://www.ibm.com/docs/en/i/7.5?topic=files-stdioh.

IBM. (07 de octubre de 2024). < stdlib.h >. https://www.ibm.com/docs/en/i/7.5?topic=files-stdlibh.

IBM. (07 de octubre de 2024). < string.h >. https://www.ibm.com/docs/en/i/7.5?topic=files-stringh.

PlantUML. (s.f). PlantUML de un vistazo. https://www.plantuml.com/plantuml/uml/SyfFKj2rKt3CoKnELR1Io4ZDoSa70000.

Sandeep. (20 de noviembre de 2021). Cómo instalar y usar «Make» en Windows. https://www.technewstoday.com/install-and-use-make-in-windows/.

Stallman, R., McGrath, R. y Smith, P. (febrero de 2023). GNU Make. https://www.gnu.org/software/make/manual/make.pdf. ISBN: 1-882114-83-3.

Yuval, A. [Yuval Adam]. (08 de mayo de 2009). *Punteros de función en C*. "Publicación en un foro en línea". Stack Overflow. https://stackoverflow.com/questions/840501/how-do-function-pointers-in-c-work.