Carátula para entrega de prácticas	
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor:	Edgar Tista García
Asignatura:	Estructuras de Datos y Algoritmos I
Grupo:	1
No. de Práctica(s):	3
Integrante(s):	Ugalde Velasco Armando
No. de Equipo de cómputo empleado:	41
No. de Lista:	
Semestre:	2020-2
Fecha de entrega:	29 de febrero de 2020
Observaciones:	

CALIFICACIÓN: _____

OBJETIVO

Utilizarás estructuras en lenguaje C para modelar tipos de datos abstractos e implementarlos en las estructuras de datos lineales.

EJEMPLOS DE LA GUÍA DE LABORATORIO

Nodo película

El objetivo del programa fue crear el modelo de la estructura película y utilizarlo, declarando e inicializando los valores correspondientes. Finalmente, se imprimieron los datos correspondientes a la estructura creada.

Sin embargo, hubo un error de compilación en un inicio. Lo anterior fue debido a un error presente en la definición de la función **llenarDatosPelicula,** cuyo tipo de dato de salida era **struct nodo**. Sin embargo, esta estructura no se encuentra definida en el programa.

A continuación, podemos observar el código de dicha función:

```
struct nodo llenarDatosPelicula(char *nombre, char *genero, short anio, short numDirectores, char *directores[10])
{
    struct pelicula movie;
    movie.nombre = nombre;
    movie.genero = genero;
    movie.anio = anio;
    movie.numDirectores = numDirectores;
    int cont = 0;
    for ( ; cont < movie.numDirectores ; cont++)
    {
        movie.directores[cont] = directores[cont];
    }
    return movie;
}
```

Como lo indica su nombre, esta función se encarga de inicializar los valores de una estructura película. Para corregir el error fue necesario modificar el tipo de dato **struct nodo** por **struct pelicula**.

```
struct pelicula matrix = llenarDatosPelicula("The matrix", "Ciencia ficción", 1999, 2, directores);
```

El error que se mostró al compilar indicaba que el problema se encontraba en esta línea. Podemos percatarnos de que tratamos de asignar un valor de tipo **struct nodo** a una variable de tipo **struct película**, lo cual es una sentencia ilegal.

A continuación, podemos observar la definición de la estructura película:

```
struct pelicula
{
    char *nombre;
    char *genero;
    short anio;
    short numDirectores;
    char *directores[10];
};
```

En la función **main**, es donde se lleva a cabo la declaración e inicialización de una variable de tipo **struct película**: *matrix,* utilizando las funciones auxiliares **llenarDatosPelicula** e **imprimirDatosPelicula**:

```
int main()
{
    char *directores[10];
    directores[0] = "Lana Wachowski";
    directores[1] = "Andy Wachowski";
    struct pelicula matrix = llenarDatosPelicula("The matrix", "Ciencia ficción", 1999, 2, directores);
    imprimirDatosPelicula(matrix);
    return 0;
}
```

La salida del programa obtenida es la siguiente:

```
PELICULA: The matrix
GENERO: Ciencia ficci¾n
ANIO: 1999
DIRECTOR(ES):
Lana Wachowski
Andy Wachowski

Process returned 0 (0x0)   execution time : 0.150 s
Press any key to continue.
```

Pila de películas

El objetivo del programa fue crear un arreglo de 2 películas. Se utilizó la misma estructura que en el programa pasado. La única excepción fue en el almacenamiento de los directores: en el primer programa se almacenaron como un arreglo de 10 punteros a carácter.

```
char directores[NUM_DIR][20];
```

En este caso se almacena un arreglo bidimensional de dimensiones 2 y 20.

Podemos observar el flujo del programa en el siguiente fragmento:

```
int main()
{
    struct pelicula arreglo[TAM];
    llenarArreglo (arreglo);
    imprimirArreglo (arreglo);
    return 0;
}
```

Primero, se declara un arreglo de películas de tamaño **TAM** (constante definida cuyo valor es 2). Después, se inicializan los dos valores del arreglo, solicitando al usuario los datos correspondientes, mediante la función **llenarArreglo**. Podemos observar el funcionamiento del programa en la siguiente captura de pantalla:

```
####### PelÝcula 1 #######
Ingrese nombre pelÝcula:dfsf
Ingrese gÚnero pelÝcula:fsdf
Ingrese a±o pelÝcula:2332
Ingrese director 1:fdsf
Ingrese director 2:fsdg
####### PelÝcula 2 #######
Ingrese nombre pelÝcula:fsdsd
Ingrese gÚnero pelÝcula:gfdss
Ingrese a±o pelÝcula:sadfd
Ingrese director 1:Ingrese director 2:sdfds
```

Finalmente, se imprimen los valores almacenados en el arreglo mediante la función **imprimirArreglo**:

```
####### Contenido del arreglo #######
####### PelÝcula 2 #######
PEL=CULA: fsdsd
G╔NERO: gfdss
AÐO: 2332
DIRECTOR(ES):
dfd
sdfds
####### PelÝcula 1 #######
PEL=CULA: dfsf
G╔NERO: fsdf
AÐO: 2332
DIRECTOR(ES):
fdsf
fsdg
```

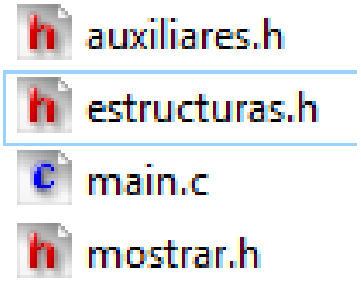
EJERCICIO

Descripción del problema

Se requiere diseñar un programa para la administración de una cadena de abarrotes. Para ello, se deben crear las estructuras Tienda, Sección, Producto, Gerente y Empleado. Además, se deben crear las funciones correspondientes para manipular las estructuras. El usuario debe ser capaz de crear al menos una tienda con 3 secciones, asignar al menos 2 productos por sección, y mostrar la información en pantalla.

Implementación

Se utilizó un enfoque ***bottom-up*** para el desarrollo del programa. Se crearon los siguientes 4 archivos de código fuente, para mantener el código legible y modularizado:



Documentos creados

Dado que se utilizó el enfoque antes mencionado, lo primero que se realizó fue la declaración de las estructuras correspondientes, de acuerdo con las definiciones proporcionadas en la descripción del problema.

estructuras.h

- Se definieron algunas macros, para lograr tener un mayor control sobre las

ocurrencias de estos fragmentos de código y seguir el principio de no repetición. Éstas se encontrarían presentes a lo largo de nuestro programa.

```
#define FLUSH fflush(stdin)
#define TAMTIENDAS 10
#define TAMSECCIONES 5
#define TAMPRODUCTOS 10
```

Macros definidas

- En cada declaración, se utilizó la palabra clave ***typedef*** para redefinir el tipo de dato de cada estructura a uno más sencillo e inteligible, con el fin de mejorar la legibilidad del código y evitar errores potenciales.

- Primero se declararon las estructuras **Gerente, Producto** y **Empleado**, las cuales no tienen como miembros a otras estructuras.

```
typedef struct gerente
{
    char nombre[100];
    char apellido[100];
    float salario;
} Gerente;


typedef struct producto
{
    int clave;
    char nombre[100];
    float precio;
} Producto;


typedef struct empleado
{
    char nombre[100];
    char apellido[100];
    char rango[100];
} Empleado;
```

Definición de estructuras

- Para el almacenamiento de cadenas, se utilizaron arreglos de 100 caracteres, en lugar de punteros a carácter.

- Finalmente, se declararon las dos estructuras restantes: **Sección** y **Tienda**. Nótese que éstas contienen como miembros a otras estructuras, las cuales ya se encuentran declaradas.

```
typedef struct seccion
{
    Empleado encargado;
    int codigo;
    char categoria[100];
    Producto productos[TAMPRODUCTOS];
    int numProductos;
} Seccion;


typedef struct tienda
{
    Gerente gerente;
    char nombre[100];
    int codigo;
    Seccion secciones[TAMSECCIONES];
    int numSecciones;
} Tienda;
```

Definición de Sección y Tienda

- Nótese que la estructura **Sección** contiene a un encargado, el cual es una estructura de tipo **Empleado**, y un arreglo de estructuras **Producto**, cuyo tamaño estaría definido por el número especificado en la macro **TAMPRODUCTOS**.

- A su vez, la estructura **Tienda** contiene a un gerente, el cual es una estructura de tipo **Empleado**, y un arreglo de estructuras **Sección**, cuyo tamaño estaría definido por el número especificado en la macro **TAMSECCIONES**.

- En ambas estructuras se agregó un miembro que especificaría el número de elementos en el arreglo correspondiente en cada estructura, con el fin de facilitar su manipulación en el programa.

Después, se realizaron las funciones correspondientes para manipular las estructuras y crear ejemplares de éstas, solicitando los datos pertinentes al usuario, presentes en el siguiente archivo.

estructuras.h

Se realizaron las siguientes funciones, propuestas en la descripción de la práctica:

```
Gerente inicializarGerente();
Producto crearProducto();
Empleado inicializarEmpleado();
Seccion crearSeccionTienda();
Tienda crearTiendaDdepartamental();
```

Prototipos

Su objetivo, como su nombre lo indica, es **inicializar o crear** un ejemplar de cada una de las estructuras. Dicha tarea se podía realizar utilizando dos enfoques: el primero, que involucraba la utilización de memoria dinámica, y el segundo, únicamente retornando una variable del tipo de estructura correspondiente, es decir, pasando por valor.

Se utilizó el segundo enfoque, ya que, al momento de la realización de la práctica, el uso de memoria dinámica era un tema aún no visto en clase. Por lo tanto, el tipo de dato de retorno de cada función, como se puede observar, corresponde con la estructura a crear.

El orden en que las funciones fueron declaradas coincide con el de la declaración de los modelos de las estructuras. Esto para enfatizar la relación que existe entre éstas: las estructuras contienen a otras estructuras, sin embargo, también era posible declararlas sin algún orden específico.

Al inicio de cada función se declaraba la variable con el tipo de estructura correspondiente, que después se retornaría.

Para capturar los datos de tipo cadena, se realizó el siguiente proceso:

```
printf("Nombre: ");
fgets(nuevoGerente.nombre, 100, stdin);
// Eliminar el salto de linea almacenado por fgets (reemplazarlo por '\0')
strtok(nuevoGerente.nombre, "\n");
FLUSH;
```

Almacenamiento de cadenas.

En este caso, se almacenó el nombre de una estructura **Gerente**. Se utilizó la función **fgets**, donde el primer parámetro especificaba la variable donde se almacenaría la cadena (**char ***). Como se mencionó anteriormente, esta variable en realidad se trataría de un arreglo de 100 caracteres. El segundo parámetro especifica la longitud de caracteres a almacenar *(100)*, y finalmente, el tercero especifica el fichero de los datos de entrada.

Sin embargo, el carácter de nueva línea **'\n'** se almacenaba de forma indeseada en

nuestra variable, por lo que fue necesario utilizar la función **strtok** para eliminarlo.

Finalmente, se utilizó la macro anteriormente definida **FLUSH**, la cual limpia el búfer de entrada para hacer posible la correcta captura de los datos.

En casos donde se almacenaban datos de tipo **int** o **float,** simplemente se utilizó la función **scanf**, donde el segundo parámetro era la dirección de memoria del miembro correspondiente de la estructura.

```
printf("Salario: ");
scanf("%f", &nuevoGerente.salario);
FLUSH;
```

Almacenamiento de dato de tipo float.

Estos procesos se realizaron en cada una de las funciones. A continuación, se presenta como ejemplo el caso particular de la función **crearProducto**:

```
Producto crearProducto()
{
    Producto nuevoProducto;
    printf("\nNUEVO PRODUCTO\n");

    printf("Clave: ");
    scanf("%d", &nuevoProducto.clave);
    FLUSH;

    printf("Nombre: ");
    fgets(nuevoProducto.nombre, 100, stdin);
    strtok(nuevoProducto.nombre, "\n");
    FLUSH;

    printf("Precio: ");
    scanf("%f", &nuevoProducto.precio);
    FLUSH;

    printf("\n");
    return nuevoProducto;
}
```

Función crearProducto.

Es importante mencionar que, en las funciones **crearSeccionTienda** y **crearTiendaDepartamental**, además de realizarse los procesos anteriores, se crean y asignan otras estructuras con ayuda de las funciones inicializadoras pertinentes. A continuación, se muestran los casos en los que esto sucede:

```
printf("** Encargado **\n");
nuevaSeccion.encargado = inicializarEmpleado();
```

Inicialización de encargado (tipo Empleado) en estructura Sección.

```
printf("** Gerente **\n");
nuevaTienda.gerente = inicializarGerente();
```

Inicialización de gerente (tipo Gerente) en estructura Tienda.

Por otro lado, dado el hecho de que los productos almacenados en cada **Sección** y las secciones almacenadas en cada **Tienda** se almacenan en arreglos, se decidió almacenar una cantidad variable de éstos, solicitando al usuario que ingrese el número de datos deseados, con el fin de que el programa sea más flexible y robusto.

Dicha tarea se llevó a cabo de la siguiente forma:

```
int numProductos = 0;

printf("Cuantos productos desea ingresar? ");
scanf("%d", &numProductos);
FLUSH;

while (numProductos < 1 || numProductos > TAMPRODUCTOS)
{
    printf("Ingrese un numero dentro del rango (1-%d): ", TAMPRODUCTOS);
    scanf("%d", &numProductos);    rodent, trodden
    FLUSH;
}

nuevaSeccion.numProductos = numProductos;

int i;
for (i = 0; i < numProductos; i++)
{
    nuevaSeccion.productos[i] = crearProducto();
}
```

Almacenamiento de productos.

En este caso, se almacenaron los productos correspondientes en la estructura Sección. Primero, se declaró e inicializó la variable **numProductos**, que almacenaría el número de productos que el usuario desea ingresar. Se solicitó el dato anterior al usuario y se

almacenó mediante el uso de la función **scanf.** Además, se utilizó un ciclo **while** para asegurar que se introdujera un número válido: que fuera un entero positivo y que no sobrepasara el tamaño del arreglo (**TAMPRODUCTOS**). Después, se almacenó el valor en el miembro correspondiente en la estructura, que posteriormente se utilizaría en las funciones para mostrar los datos. Finalmente, se solicitó y almacenó la cantidad de productos especificada mediante un ciclo **for,** utilizando la función **crearProducto** para asignar a cada elemento del arreglo de productos el ejemplar correspondiente.

En la estructura se llevó a cabo exactamente el mismo proceso, pero, en este caso, el arreglo fue de estructuras **Sección**:

```
int numSecciones = 0;

printf("Cuantas secciones desea ingresar? ");
scanf("%d", &numSecciones);
FLUSH;

while (numSecciones < 1 || numSecciones > TAMSECCIONES)
{
    printf("Ingrese un numero dentro del rango (1-%d): ", TAMSECCIONES);
    scanf("%d", &numSecciones);
    FLUSH;;
}

nuevaTienda.numSecciones = numSecciones;

int i;
for (i = 0; i < numSecciones; i++)
{
    nuevaTienda.secciones[i] = crearSeccionTienda();
}
```

Almacenamiento de secciones.

Después, se crearon las funciones utilizadas para mostrar los datos de las estructuras.

mostrar.h

En este documento se encuentran las siguientes funciones:

```
void mostrarTiendasDep(Tienda tiendas[TAMTIENDAS], int numTiendas);
void mostrarSecciones(Seccion secciones[TAMSECCIONES], int numSecciones);
void mostrarProductos(Producto productos[TAMPRODUCTOS], int numProductos);
void mostrarGerente(Gerente gerente);
```

Prototipos

Como su nombre lo indica, su objetivo es imprimir los datos correspondientes a cada miembro de la estructura correspondiente. En este caso, se declaró primero la función

mostrarTiendasDep, la cual implica la ejecución de las funciones **mostrarSecciones** y **mostrarGerente**. A su vez, dentro de la función **mostrarSecciones** se ejecuta la función **mostrarProductos.** Lo anterior es consecuencia de la presencia de estructuras dentro de la definición de otras estructuras.

Podemos notar que en las primeras tres funciones se utilizan dos parámetros: el arreglo correspondiente de estructuras y el número de elementos que ahí se encuentran.

En el primer caso, la función **mostrarTiendasDep** se ejecuta en la función **main**: el primer argumento es un arreglo de **Tiendas** y el segundo el **número** de Tiendas creadas.

En el caso de las funciones **mostrarSecciones** y **mostrarProductos**, el segundo argumento es el tamaño especificado en la estructura que contiene el arreglo. Es decir, en la estructura Tienda se encuentra el arreglo de secciones, junto con el número de secciones inicializadas. De la misma forma, en la estructura Sección se encuentra el arreglo de Productos, junto con el número de productos inicializados.

Lo anterior se realiza para facilitar la impresión de los datos correspondientes mediante el uso de ciclos **for**, donde el límite superior es el número de ejemplares inicializados en cada arreglo. A continuación, se muestra claramente lo anterior:

```
for (i = 0; i < numTiendas; i++)
{
    printf("\n\n**** TIENDA %s ****\n\n", tiendas[i].nombre);
    mostrarGerente(tiendas[i].gerente);
    printf("Codigo de la tienda: %d\n", tiendas[i].codigo);
    printf("\n*** SECCIONES ***\n");
    mostrarSecciones(tiendas[i].secciones, tiendas[i].numSecciones);
}
```

Ciclo for en la función mostrarTiendasDep.

En la captura anterior, se puede observar que se imprime cada elemento de cada estructura **Tienda** presente en el arreglo proporcionado. Se utilizan las funciones **mostrarGerente** y **mostrarSecciones** para realizar las tareas correspondientes. El segundo parámetro de **mostrarSecciones** es el número de secciones presentes en la **Tienda** correspondiente.

Asimismo, como ya se mencionó, en las funciones restantes (a excepción de **mostrarGerente**) se utilizaron los ciclos de la misma forma:

```
for (i = 0; i < numSecciones; i++)
{
    printf("\n   SECCION %d   \n", i+1);
    printf("Encargado: %s %s (Rango %s)\n", secciones[i].encargado.nombre, s
    printf("Codigo: %d\n", secciones[i].codigo);
    printf("Categoria: %s\n", secciones[i].categoria);
    printf("\nPRODUCTOS DE LA SECCION");
    mostrarProductos(secciones[i].productos, secciones[i].numProductos);
}
```

Ciclo presente en la función mostrarSecciones

```
for (i = 0; i < numProductos; i++)
{
    printf("\n\nProducto %d\n", i+1);
    printf("Nombre: %s\n", productos[i].nombre);
    printf("Clave: %d\n", productos[i].clave);
    printf("Precio: %f\n\n", productos[i].precio);
}
```

Ciclo presente en la función mostrarProductos

Finalmente, el flujo del programa se encuentra definido en la función **main.**

main.c

En este documento se encuentra únicamente la función main.

```
Tienda tiendas[TAMTIENDAS];
int bandera = 1;
int noTiendas = 0;
```

Declaración e inicialización de variables

Primero, se declaró el arreglo de estructuras **Tienda** con el tamaño definido en la macro **TAMTIENDAS**, para almacenar las tiendas que el usuario desee crear. Después, la variable **bandera** se inicializó a 1, un valor que se evalúa como verdadero al momento de ejecutar el ciclo. La variable **noTiendas** llevaría el registro de las tiendas que el usuario ha creado; su valor inicial obviamente es 0.

```
while(bandera)
{
    printf("  MENU PRINCIPAL\n");
    printf("1) Crear una tienda (%d restantes)\n2) Mostrar tiendas\n3) Salir\n", TAMTIENDAS - noTiendas);

    int opcion;
    scanf("%d", &opcion);
    FLUSH;
```

Menú principal

Después, la ejecución del programa se realizaría dentro de un ciclo **while,** para continuar ejecutándose hasta que el usuario decidiera salir. Se muestra el menú principal con 3 opciones, y se solicita al usuario que ingrese la opción deseada. Posteriormente, se ejecutaría la función correspondiente mediante la utilización de una sentencia **switch**. Las opciones consisten en lo siguiente:

1. **Crear una tienda**.

   En esta opción también se muestra el número de tiendas disponibles para su almacenamiento, dependiendo del tamaño del arreglo de Tiendas definido por **TAMTIENDAS,** y de las Tiendas creadas por el usuario. Si el número de tiendas restantes disponibles es 0, le será notificado al usuario y no podrá crear una nueva Tienda.

   Si aún hay espacio disponible, se ejecuta la función **crearTiendaDepartamental**, para inicializar los valores correspondientes y se asigna el valor retornado a la posición del arreglo de Tiendas adecuada.

   También se realiza el incremento del contador del número de tiendas. En la siguiente captura de pantalla se observa claramente el flujo mencionado:

```
if (noTiendas < TAMTIENDAS)
{
    tiendas[noTiendas++] = crearTiendaDepartamental();
}
else
{
    printf("Se ha agotado el numero posible de tiendas\n");
}
```

Opción 1

2. **Mostrar tiendas**

   En esta opción únicamente se ejecuta la función **mostrarTiendasDep**, que toma como argumentos el **arreglo** de Tiendas y el **número** de Tiendas creadas.

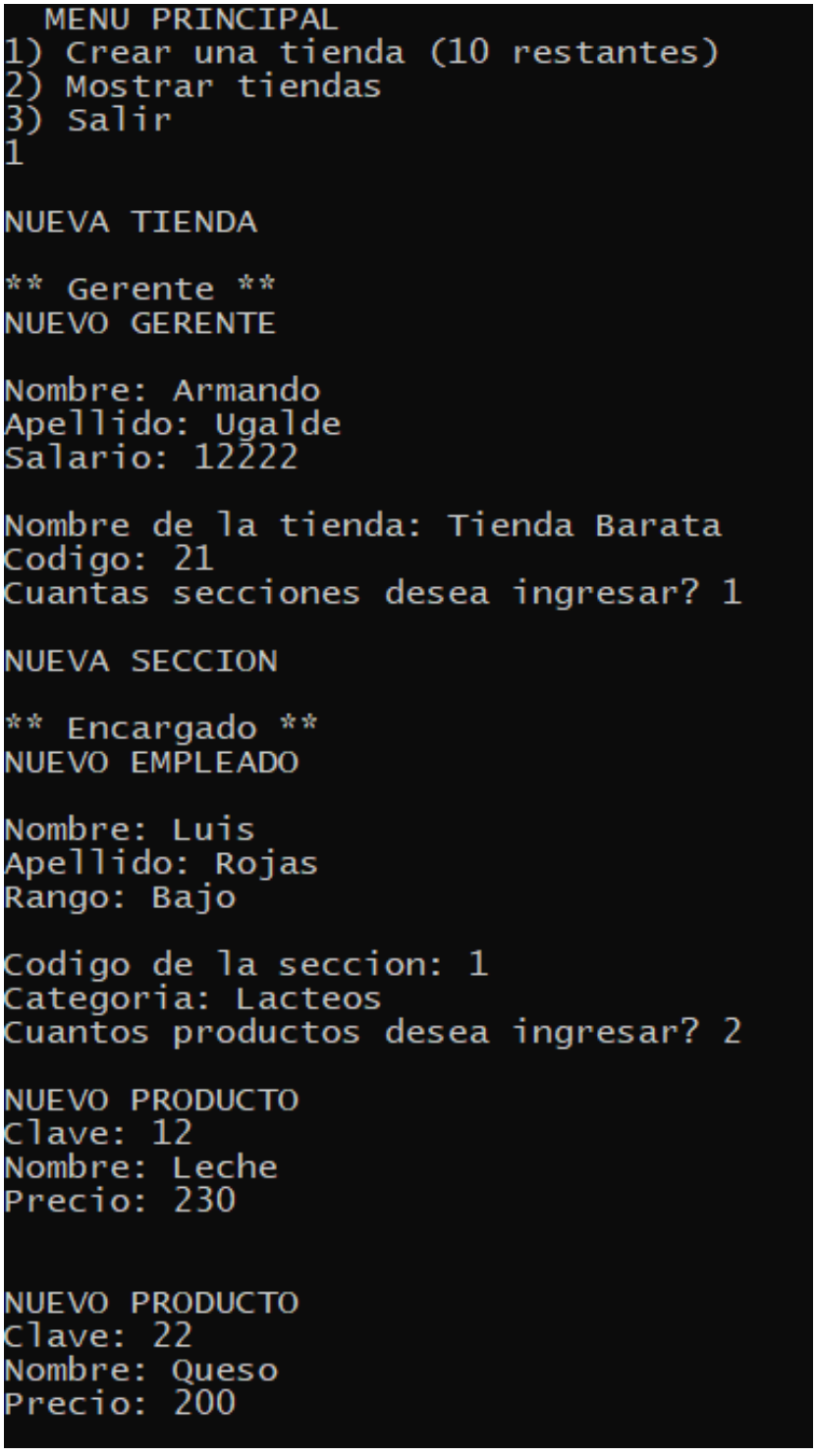
```
mostrarTiendasDep(tiendas, noTiendas);
```

Función mostrarTiendasDep

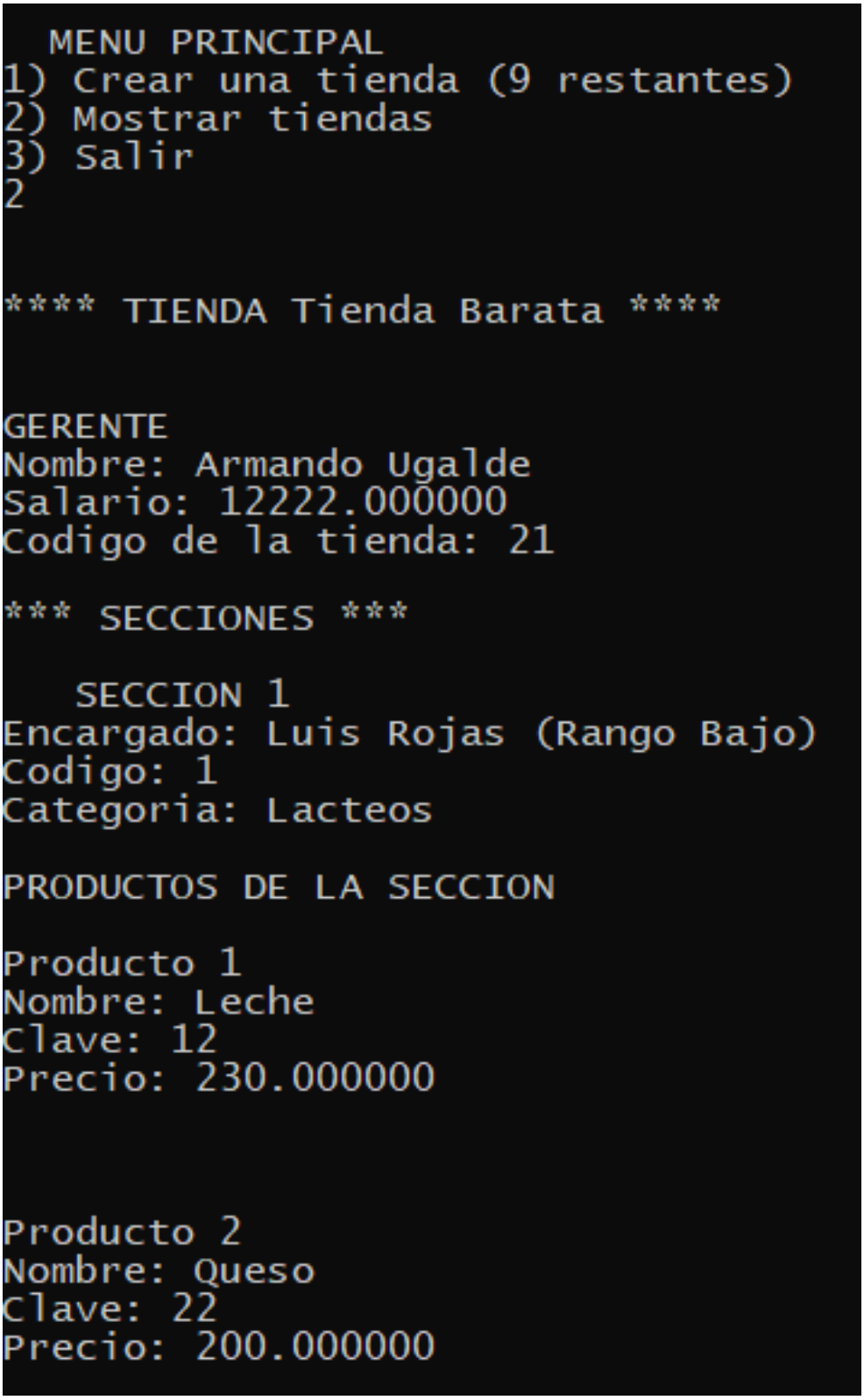
3. **Salir**

   En esta función se cambia el valor de la variable **bandera** por 0, por lo que, al ser un valor que se evalúa a falso, el ciclo se detendría, y, en consecuencia, el programa terminaría.

A continuación, se muestra un ejemplo de la utilización del programa:



Creación de tienda



Mostrar tiendas

CONCLUSIONES

Podemos decir que se cumplió el objetivo planteado inicialmente, ya que se modelaron los tipos de datos abstractos pertinentes, es decir, **Tienda, Sección, Producto, Gerente y Empleado**, implementándolos mediante el uso de estructuras. Además, se crearon las funciones adecuadas para su manipulación y utilización, lo cual se considera una buena práctica de programación, al contribuir a la modularización del código.

Cabe mencionar que se utilizó la sentencia **typedef** para mejorar la legibilidad del código.

El problema planteado me pareció adecuado para la utilización de los conceptos planteados. Además, se tuvo libertad para escoger el enfoque mediante el que se resolvería el problema, lo cual me parece ideal. El programa, en este caso, se realizó utilizando el paradigma **bottom-up**, es decir, se empezó con la implementación de los componentes más elementales, y se desarrollaron los componentes más complejos al final.