RECUPERATORIO PARCIAL 1

MATERIA: LABORATORIO 1

DIVISION: 1F

ALUMNO: ALFREDO GONZALEZ

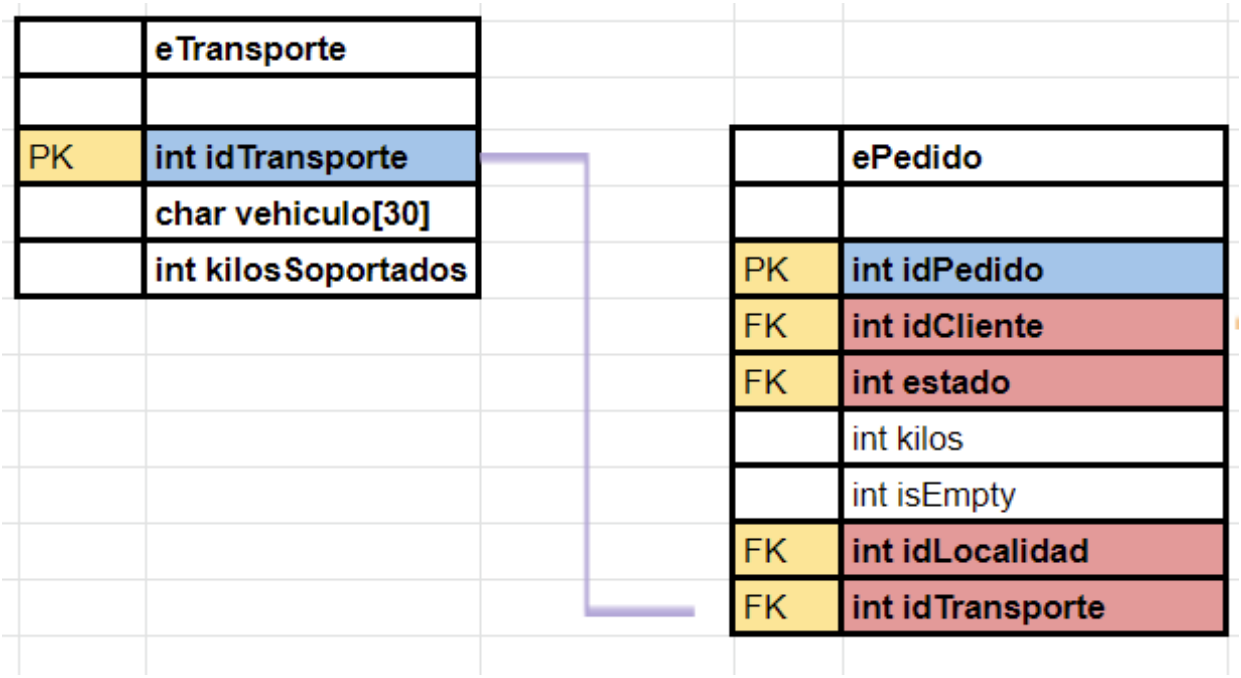
PROFESOR: GERMÁN SCARAFILO

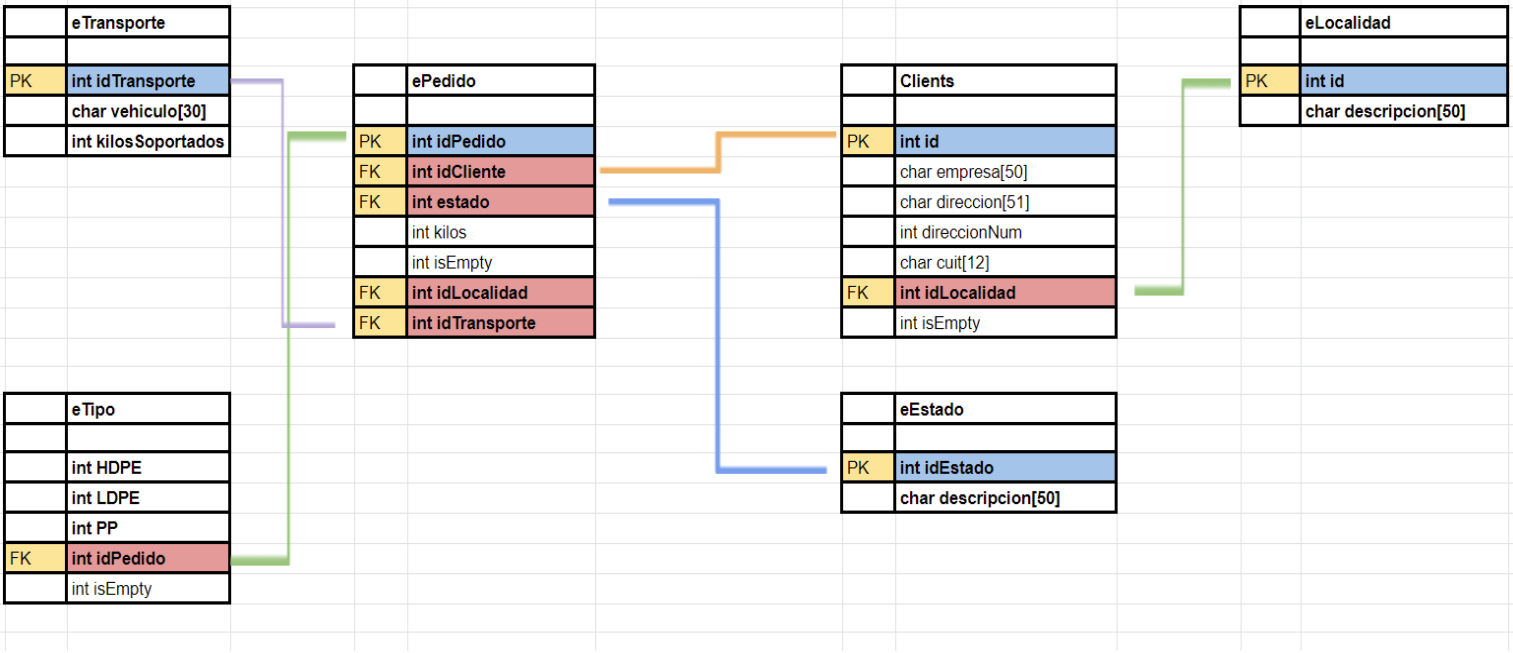
27/11/2021

3.B)

La estructura agregada se trata de eTransporte, al cual va darle al cliente 3 opciones de transporte según la cantidad de kilos del pedido, la misma va a contar con un idTransporte para cada uno de los 3 transportes, la descripción de los mismos que en este caso se llama vehículo y la cantidad de kilos que soportan según el vehículo.

El idTransporte va a ser pedido al usuario al momento de hacer la carga de un pedido, la estructura ePedido va tener su propio idTransporte como una Foreign Key, y esa variable va a ser comparada mediante una función de eTransporte para cargar la descripción de la misma, todo esto se va a ver en el código.




eTransporte
PK int idTransporte
char vehiculo[30]
int kilosSoportados
ePedido
PK int idPedido
FK int idCliente
FK int estado
int kilos
int isEmpty
FK int idLocalidad
FK int idTransporte
Clients
PK int id
char empresa[50]
char direccion[51]
int direccionNum
char cuit[12]
FK int idLocalidad
int isEmpty
eLocalidad
PK int id
char descripcion[50]
eTipo
int HDPE
int LDPE
int PP
FK int idPedido
int isEmpty
eEstado
PK int idEstado
char descripcion[50]

3.C)

ENUNCIADO:

```
459 int primerClienteMoto(ePedido* lista, Clients* list, eTransporte* listaTransporte, int len, int tam, int tamTransporte)
460 {
461     int allOk=0;
462     int flag =1;
463     char primerClienteMoto[50];
464     char cuitPrimerMoto[20];
465 
466     if(lista != NULL && list != NULL && listaTransporte != NULL && len > 0 && tam > 0 && tamTransporte > 0)
467     {
468         for(int i=0; i<len;i++)
469         {
470             for(int j=0; j<tam;j++)
471             {
472                 for(int k=0; k<tamTransporte;k++)
473                 {
474                     if(lista[j].idCliente == list[i].id && lista[j].isEmpty == CARGADO && lista[j].estado == 2
475                             && listaTransporte[k].idTransporte == 1 && lista[j].idTransporte == 1 && flag == 1 )
476                     {
477                             flag = 0;
478                             strcpy(primerClienteMoto, list[i].empresa);
479                             strcpy(cuitPrimerMoto, list[i].cuit);
480                             printf("El primer cliente con envio entregado en moto es: \nNOMBRE:%s\nCUIT:%s\n", primerClienteMoto, cuitPrimerMoto);
481                             break;
482                     }
483                 }
484             }
485         }
486         allOk = 1;
487         if(flag == 1)
488         {
489             printf("Error.. no hay pedidos entregados en moto\n");
490         }
491     }
492     return allOk;
493 }
494 
```

El filtro complejo elegido es el de la función primerClienteMoto, que lo que hace es por medio de las listas de pedidos, listas de clientes y lista de transportes, recorrer el array de pedidos primero, luego el array de clientes y por el último el de transportes, buscando la primera ocurrencia de cliente con un pedido **COMPLETADO** y el id del transporte sea el id de **MOTO**, cuando el programa encuentra esto, va a setear una flag en 0, lo cual va a hacer que no vuelva a entrar al if nuevamente y printea el nombre y el cuit de la persona que tuvo el primer pedido completado en moto. Así mismo, si no hay pedidos completados, la flag estado se ocuparía de que se imprima el error correspondiente en pantalla.

La función retorna 0 en caso de error y 1 en caso de que las listas sean distinto de null y el tamaño sea menor al de las listas.

```c
int vehiculoMasKilosEnviados(ePedido* lista, int len)
{
    int allOk= 0;
    int acumMoto=0;
    int acumAuto=0;
    int acumCamion=0;
    if(lista != NULL && len >0)
    {
        for(int i=0; i<len; i++)
        {

            if(lista[i].isEmpty == CARGADO && lista[i].estado == COMPLETADO)
            {
                switch(lista[i].idTransporte)
                {
                    case 1:
                        acumMoto = lista[i].kilos + acumMoto;
                    break;
                    case 2:
                        acumAuto = lista[i].kilos +acumAuto;
                    break;
                    case 3:
                        acumCamion = lista[i].kilos + acumCamion;
                    break;
                }
            }
        }
        allOk = 1;
        printf("---------Vehiculo con mas kilos enviados---------\n");
        if(acumMoto > acumAuto && acumMoto > acumCamion)
        {
            printf("El vehiculo con mas kilos enviados fue moto con un total de: [%d]\n", acumMoto);
        }
        else
        {
            if(acumAuto > acumMoto && acumAuto > acumCamion)
            {
                printf("El vehiculo con mas kilos enviados fue auto con un total de: [%d]\n", acumAuto);
            }
            else
            {
                if(acumCamion > acumAuto && acumCamion > acumMoto)
                {
                    printf("El vehiculo con mas kilos enviados fue camion con un total de: [%d]\n", acumCamion);
                }
                else
                {
                    printf("No se pudo encontrar el vehiculo con mas envios cargados\n");
                }
            }
        }
    }

    return allOk;
}
```

Esta función se trataría del segundo filtro, que solo recibe como parámetro la lista de pedidos con su tamaño, se encarga de por medio de acumuladores, ir llenando variables con el valor de kilos enviados por pedido según el vehículo, para luego mostrar en pantalla el vehículo que más kilos envió. A sí mismo cuenta con validación en caso de que no haya un vehículo con más cantidad de envíos y también en el main, tiene la validación de que tiene que haber pedidos enviados.

retorna 0 si hubo error en la listaPedidos o en el tamaño de la lista, y retorna 1 si pudo acceder a la lista correctamente

3.D)

Clientes

```c
typedef struct
{
    int id;
    char empresa[51];
    char direccion[51];
    int direccionNum;
    char cuit[12];
    int idLocalidad;
    int isEmpty;
}Clients;
```

```c
/// @fn int IniciarClientes(Clients*, int)
/// @brief recorre el array de lista de clientes dejando todos los espacios vacios
/// @param list
/// @param len
/// @return 0 despues de recorrer el array
int IniciarClientes(Clients* list, int len);
/// @fn int BuscarLibre(Clients*, int)
/// @brief recorre el array de lista de clientes hasta encontrar el primer indice vacio y lo guarda.
/// @param list
/// @param len
/// @return el indice vacio encontrar
int BuscarLibre(Clients* list, int len);
/// @fn int AgregarCliente(Clients*, int*, int)
/// @brief llama a BuscarLibre y guarda el indice vacio encontrado, en ese indice pide y valida los datos personales del usuario
/// @param list
/// @param pId
/// @param len
/// @return retorna 1 si pudo acceder a la lista y 0 si hubo algun error
int AgregarCliente(Clients *list, eLocalidad* localidades, int *pId,int len);
/// @fn int EncontrarClientePorID(Clients*, int, int)
/// @brief pasa un id por parametro y lo compara con todos los id de la lista de clientes, al encontrar el id, guarda el indice.
/// @param list
/// @param len
/// @param id
/// @return retorna el indice del id buscado, o -1 si no pudo encontrarlo
int EncontrarClientePorID(Clients* list, int len,int id);
/// @fn int ModificarCliente(Clients*, int)
/// @brief pide un id al usuario, luego con el id llama a la funcion EncontrarClientePorID para encontrar el indice del cliente, una vez
/// accede a los datos da al cliente la opcion de cambiar la direccion o la localidad.
/// @param list
/// @param len
/// @return retorna 0 si tuvo algun error y 1 si pudo acceder correctamente.
int ModificarCliente(Clients* list, int len, eLocalidad* localidades);
/// @fn int BajaCliente(Clients*, int, int)
/// @brief recibe un id como parametro, llama a la funcion EncontrarClientePorID y lo guarda en un indice, al acceder al indice da opcion al usuario
/// de eliminarlo o cancelar la baja. Si la baja se realiza, deja la direccion de ese cliente en 0
/// @param list
/// @param len
/// @param id
/// @return retorna -1 si hubo algun error y 0 si pudo acceder correctamente
int BajaCliente(Clients* list, int len, int id);
```

```
/// @fn int ContadorClientes(Clients*, int*, int)
/// @brief recibe un puntero de int como parametro, recorre el array de clientes y por cada uno que este cargado suma 1 al contador
/// @param list
/// @param contadorClientes
/// @param len
/// @return -1 si hubo algun error y 0 si pudo acceder correctamente
int ContadorClientes(Clients* list, int *contadorClientes, int len);
/// @fn void MostrarCliente(Clients, eLocalidad*, int)
/// @brief recibe como parametro los clientes, la lista de localidades y el tamaño de la lista de localidades.
/// Llama a CargarDescripcionLocalidad y si no tiene ningun error muestra el cliente junto con la localidad
/// @param x
/// @param localidades
/// @param tamLoc
void MostrarCliente(Clients x, eLocalidad* localidades, int tamLoc);
/// @fn int ImprimirClientes(Clients*, eLocalidad*, int, int)
/// @brief recibe como parametro los clientes, la lista de localidades y el tamaño de la lista de localidades y de clientes. recorre el array de clientes y muestra todos los que esten cargados
/// @param list
/// @param localidades
/// @param tamLoc
/// @param len
/// @return -1 si hubo error y 0 si pudo acceder correctamente
int ImprimirClientes(Clients* list,eLocalidad* localidades, int tamLoc, int len);
```

Pedidos

```
typedef struct
{
    int idCliente;
    int idPedido;
    int estado;
    int kilos;
    int isEmpty;
    int idLocalidad;
    int idTransporte;
}ePedido;
```

```
/// @brief inicializa la lista de pedidos y las deja todas en vacio
/// @param list
/// @param len
/// @return -1 si hubo error y 0 si pudo funcionar correctamente
int IniciarPedidos(ePedido *list, int len);
/// @fn int BuscarLibrePedidos(ePedido*, int)
/// @brief recorre el array y guarda la primera direccion vacio en un indice
/// @param list
/// @param len
/// @return -1 si hubo error y el indice si pudo acceder correctamente
int BuscarLibrePedidos(ePedido *list, int len);
/// @fn int AgregarPedido(ePedido*, eEstado*, int, int, int, int*)
/// @brief busca el indice libre, pide y valida los datos personales del usuario, por ultimo cambia el estado del pedido a pendiente
/// @param list
/// @param estados
/// @param tamEstado
/// @param id
/// @param len
/// @param idPedido
/// @return -1 si hubo error y 0 si funciona correctamente
int AgregarPedido(ePedido* list, eEstado* estados, Clients* lista, eTransporte* listaTransporte, int tamTransporte, int tamEstado, int id, int len, int *idPedido);
/// @fn void MostrarPedido(ePedido, eEstado*, int)
/// @brief Muestra los pedidos tanto en pendiente como en completado junto con los datos de un cliente
/// @param x
/// @param estados
/// @param tamEstado
void MostrarPedido(ePedido x, eEstado* estados, eTransporte* listaTransporte, int tamEstado, int tamTransporte);
/// @fn int ImprimirPedidos(ePedido*, eEstado*, int, int)
/// @brief llama a mostrar pedidos e imprime todos los pedidos junto con el id de los clientes
/// @param list
/// @param estados
/// @param tamEstado
/// @param len
/// @return -1 si hubo error y 0 si pudo funcionar correctamente
int ImprimirPedidos(ePedido* list, eEstado* estados, eTransporte* listaTransporte, int tamEstado, int tamTransporte, int len);
int BuscarPedido(ePedido* list, int len, int id);
/// @fn int BuscarPedidoPorID(ePedido*, int, int)
/// @brief recibe un id como parametro y recorre el array en busca del pedido de ese id, cuando lo encuentra guarad el indice
/// @param list
/// @param len
/// @param id
/// @return -1 si hubo error y el indice si funciono correctamente
int BuscarPedidoPorID(ePedido* list, int len, int id);
```

```
/// @fn int CantidadPedidosPorID(ePedido*, int, int)
/// @brief busca por un id en la lista de pedidos el indice, al matchear agrega 1 al contador de pedidos
/// @param list
/// @param len
/// @param id
/// @return -1 si hubo error y los pedidos encontrados si funciono correctamente
int CantidadPedidosPorID(ePedido* list, int len, int id);
/// @fn int ContadorPedidos(ePedido*, int*, int, int)
/// @brief recorre el array de pedidos, al encontrar matchs con los ids y cuando las listas estan cargadas y pendientes, agrega 1 al contador de pedidos
/// @param list
/// @param contadorPedidos
/// @param len
/// @param id
/// @return -1 si hubo error y 0 si pudo funcionar
int ContadorPedidos(ePedido* list, int *contadorPedidos, int len, int id);
/// @fn int PedidosPorLocalidad(ePedido*, eLocalidad*, int, int, int)
/// @brief compara el id que pasa por parametro junto con el id de localidad y el estado del pedidos, al hacer match carga 1 al contador de pedidos
/// @param lista
/// @param list
/// @param tam
/// @param len
/// @param id
/// @return -1 si hubo error y el contador de pedidos si funciono correctamente
int PedidosPorLocalidad(ePedido* lista, eLocalidad* list, int tam, int len, int id);
/// @fn int ImprimirClientePendiente(ePedido*, Clients*, int)
/// @brief recorre la listas de pedidos y la lista de clientes, compara los ids de la lista y valida que el estado del pedido sea en pendiente, luego imprime los datos del clien
/// @param lista
/// @param list
/// @param len
/// @return -1 si hubo error y 0 si pudo funcionar correctamente
int ImprimirClientePendiente(ePedido* lista, Clients* list, int len, int tam);
/// @fn void MostrarPedidoPendiente(Clients*, ePedido*)
/// @brief compara los id del pedido junto con lo del cliente y muestra los datos del cliente al hacer match
/// @param list
/// @param lista
void MostrarPedidoPendiente(Clients* list, ePedido* lista);
/// @fn int ImprimirPedidosPendientes(ePedido*, Clients*, int)
/// @brief  recorre los array de cliente y de pedidos, verifica que el estado del pedido sea pendiente y por ultimo imprime todos los pedidos en pendiente junto con info del cli
/// @param lista
/// @param list
/// @param len
/// @return -1 si da error y 0 si funciono correctamente
int ImprimirPedidosPendientes(ePedido* lista, Clients* list, int len, int tam);
```

```
/// @fn int vehiculoMasKilosEnviados(ePedido*, int)
/// @brief Recorre el array en busca del vehiculo que mas cantidad de kilos envio en el transcurso del programa y lo printea
/// @param lista
/// @param len
/// @return 0 si da error y 1 si funciona correctamente
int vehiculoMasKilosEnviados(ePedido* lista, int len);
/// @fn int primerClienteMoto(ePedido*, Clients*, eTransporte*, int, int, int)
/// @brief recorre los array de las 3 listas en busca del primer envio completado que haya sido realizado en moto y printea el nombre y el cuil de
/// @param lista
/// @param list
/// @param listaTransporte
/// @param len
/// @param tam
/// @param tamTransporte
/// @return 0 si da error y 1 si funciona correctamente
int primerClienteMoto(ePedido* lista, Clients* list, eTransporte* listaTransporte, int len, int tam, int tamTransporte);
#endif /* PEDIDOS_H_ */
```

Localidad:

```c
typedef struct
{
    int id;
    char descripcion[50];
}eLocalidad;

/// @fn int mostrarLocalidades(eLocalidad*, int)
/// @brief recibe como parametro las localidades y el tamaño de la lista. Recorre la lista de localidades y muestra el id de las localidades junto con la descripcion
/// @param localidades
/// @param tam
/// @return 0 si hubo error y 1 si pudo acceder correctamente
int mostrarLocalidades(eLocalidad* localidades, int tam);
/// @fn int cargarDescripcionLocalidad(eLocalidad*, int, int, char[])
/// @brief recibe el idLocalidad como parametro y recorre el array de localidades mientras compara el id ingresado con el de la estructura.
/// Al matchear copia la descripcion de la estructura con la descripcion recibida por parametro
/// @param localidades
/// @param tam
/// @param idLocalidad
/// @param descripcion
/// @return -1 si hubo error y 0 si pudo acceder correctamente
int cargarDescripcionLocalidad(eLocalidad* localidades, int tam,int idLocalidad, char descripcion[]);

#endif /* LOCALIDAD_H_ */
```

Transporte:

```c
typedef struct
{
    int idTransporte;
    char vehiculo[30];
    int kilosSoportados;
}eTransporte;

#define TRANSPORTE_H_

/// @fn int mostrarTransportes(eTransporte*, int)
/// @brief Muestra la lista de transportes y los kilos correspondientes que soporta cada uno
/// @param listaTransporte
/// @param tamTransporte
/// @return 0 si da error y 1 si funciona correctamente
int mostrarTransportes(eTransporte* listaTransporte, int tamTransporte);
/// @fn int cargarDescripcionTransporte(eTransporte*, int, int, char[])
/// @brief recibe el id de un vehiculo como parametro y busca matchear con el id de la estructura. Al matchear, copa la descripcion de dicho vehiculo en una variable
/// @param listaTransporte
/// @param tamTransporte
/// @param idVehiculo
/// @param descripcion
/// @return 0 si da error y 1 si funciona correctamente
int cargarDescripcionTransporte(eTransporte* listaTransporte, int tamTransporte,int idVehiculo, char descripcion[]);

#endif /* TRANSPORTE_H_ */
```

Estado:

```
typedef struct
{
    int idEstado;
    char descripcion[50];
}eEstado;

/// @fn int MostrarEstados(eEstado*, int)
/// @brief recibe como parametro la lista de estado junto con el tamaño de la lista, recorre la lista y muestra el id del estado junto con la descripcion
/// @param estados
/// @param tamEstado
/// @return 0 si hubo error y 1 si pudo funcionar correctamente
int MostrarEstados(eEstado* estados, int tamEstado);
/// @fn int CargarDescripcionEstado(eEstado*, int, int, char[])
/// @brief recibe como parametro la lista de estado junto con el tamaño de la lista,
/// recorre la lista y compara el idEstado que recibe como parametro con el idEstado de la estructura, al matchear copia la descripcion de ese id en una variable
/// @param estados
/// @param tamEstado
/// @param idEstado
/// @param descripcion
/// @return -1 si hubo error y 0 si pudo funcionar correctamente
int CargarDescripcionEstado(eEstado* estados, int tamEstado, int idEstado, char descripcion[]);


```

Tipos

```
#include "Pedidos.h"
#include "ArrayClients.h"

typedef struct
{
    int HDPE;
    int LDPE;
    int PP;
    int idPedido;
    int isEmpty;
}eTipo;
```

```c
/// @fn int IniciarTipos(eTipo*, int)
/// @brief recorre el array de tipo dejando todos los espacios vacios
/// @param lista
/// @param len
/// @return -1 si error y 0 si pudo funcionar correctamente
int IniciarTipos(eTipo *lista, int len);
/// @fn int BuscarLibreTipo(eTipo*, int)
/// @brief recorre el array de tipo hasta encontrar indice vacio y lo guarda
/// @param lista
/// @param len
/// @return el indice encontrado previamente
int BuscarLibreTipo(eTipo* lista, int len);
/// @fn int PedirTipos(ePedido*, int, int, eTipo*, int*, int*)
/// @brief recibe como parametro la lista de pedidos y 2 punteros de int, uno de acumulador y otro de contador de clientes.
/// Busca el indice deseado por medio de un ID, al encontrar el indice lo compara con el del estado del pedido y
/// si hay match pide al usuario los tipos de plasticos para su pedido
/// @param list
/// @param len
/// @param id
/// @param lista
/// @param pAcum
/// @param pClientes
/// @return 0 si hubo error y 1 si pudo funcionar correctamente
int PedirTipos(ePedido* list, int len, int id, eTipo* lista,int *pAcum, int *pClientes);
/// @fn int ImprimirPedidosCompletados(ePedido*, Clients*, int, eTipo*)
/// @brief recorre el array de pedidos y de tipos. Compara el estado del pedido, los ids del pedido y verifica que ambas listas esten cargadas,
/// por ultimo imprime los pedidos completados con los datos del cliente junto con los tipos de plasticos del pedido
/// @param lista
/// @param list
/// @param len
/// @param listaT
/// @return -1 si hubo error y 0 si pudo funcionar correctamente
int ImprimirPedidosCompletados(ePedido* lista, Clients* list, int len, eTipo* listaT);
/// @fn void calcularPromedio(ePedido*, int, int*, int*)
/// @brief calcula el promedio de PP que fue entregado en el transcurso del programa y lo printea
/// @param lista
/// @param len
/// @param pAcum
/// @param pContador
void calcularPromedio(ePedido* lista, int len, int *pAcum, int* pContador);
```

VIDEO DEFENSA:

OPCION 1:

https://drive.google.com/file/d/1cd3UFCi8pbj6g__kR_-6PRyWz97T1-_L/view?usp=sharing

OPCION 2(RESPALDO): https://youtu.be/kDU8EeqrCQQ