



UNIVERSIDAD NACIONAL EXPERIMENTAL DE
GUAYANA
VICE RECTORADO ACÁDEMICO
COORDINACION DE INGENIERIA EN
INFORMATICA

TECNICAS DE PROGRAMACION

II

SECCIÓN: 02
PROF. AIMARA COVA

Sistema de Recomendación de Productos Personalizados

Integrantes:

Cesar Rivera

Isabella Español

Enunciado del Proyecto: Sistema de Recomendación de Productos Personalizados.

Objetivo: Desarrollar un sistema de recomendación de productos que ofrezca sugerencias personalizadas a los usuarios calculando en sus preferencias, su historial de interacciones y un catálogo de productos, haciendo uso exclusivo de las siguientes estructuras de datos: struct, arrays (o matrices), pilas, colas, listas enlazadas (simples o dobles) y árboles binarios.

Funcionalidades requeridas:

1. Gestión de Perfiles de Usuario

El sistema debe permitir la creación y gestión de perfiles de usuario. Cada perfil de usuario deberá almacenar la siguiente información:

- Nombre
- Apellido
- ID
- Usuario (para el inicio de sesión)
- Contraseña
- Preferencias de Productos: Un conjunto de categorías y marcas que el usuario ha indicado como de su interés.

● Historial de Interacciones: Un registro de los productos que el usuario ha comprado o simplemente ha visualizado en la plataforma.

2. Catálogo y Gestión de Productos

El sistema debe mantener un catálogo de productos organizado por categorías. Cada producto debe incluir la siguiente información:

- Descripción
- ID
- Marca
- Precio
- Calidad del Producto (variable que maneje la calidad, por ejemplo, un entero del 1 al 5)
- Estructura de datos : usted deberá especificar las estructuras de datos adecuadas para manejar el catálogo y gestión de productos.

3. Funcionalidades de Búsqueda de Productos

El sistema debe permitir a los usuarios buscar productos de diversas maneras:

- Por Categoría: Listar todos los productos pertenecientes a una categoría específica.
- Por Rango de Precios: Listar productos cuyo precio se encuentre dentro de un rango definido.
- Por Marca: Listar todos los productos de una marca específica.
- Por Descripción: Buscar productos que contengan una palabra o frase específica en su descripción.

4. Sistema de Recomendación Personalizado

Esta es la funcionalidad central del sistema. Basándose en las preferencias del usuario y su historial de interacciones, el sistema debe generar recomendaciones de productos

- Recomendaciones por Marcas Preferidas: Si el usuario tiene preferencias por algunas marcas, el sistema deberá presentarle recomendaciones de productos de esas marcas, incluso si no pertenecen a las categorías que el usuario ha marcado como preferidas . Las recomendaciones deben priorizar estas marcas.
- Recomendaciones por Historial de Interacciones: El sistema debe recomendar productos similares a los que el usuario ha comprado o visto previamente. La similitud se puede determinar por:
 - Misma Categoría: Otros productos de la misma categoría que los productos en el historial.
 - Misma Marca: Otros productos de la misma marca que los productos en el historial.
 - Calidad Similar: Productos con una calificación de calidad comparable.

5. Administración de memoria

- Se debe implementar la asignación dinámica de memoria para la creación de perfiles de usuario, el almacenamiento del historial de interacciones, así como para la adición de productos al catálogo y la generación de las listas de recomendaciones. Se espera un

manejo adecuado de la memoria (asignación y liberación) para evitar fugas.

6. Uso de Recursión

- Se sugiere la aplicación de recursión en algoritmos de búsqueda dentro del árbol de perfiles de usuario, la navegación por las listas de categorías/productos, o en la generación de recomendaciones de que puedan implicar patrones anidados en las preferencias, el historial o el catálogo.

7. Conversión de tipos

- Se deberá considerar la conversión entre diferentes tipos de datos cuando sea necesario, especialmente para el análisis de datos (ej. cadenas a números para comparaciones, etc.).

Análisis del problema

Entrada	Proceso	Salida
Datos personales del usuario: String Nombre String Apellido Int ID String Usuario String Contraseña Puntero Preferencia Puntero Historial de interacciones	Validar y almacenar perfiles en árbol binario por usuario. Gestionar preferencias en listas Enlazadas simples. Registrar interacciones en listas o colas FIFO.	Imprimir: Perfil creado exitosamente.
Productos del catalogo: String Descripcion Int ID String Marca Int precio String Calidad.	Almacenar productos en matrices	

EL catalogo.cpp es donde se implementa todas las definiciones previas en el catalogo.h que es un modo de function.h archivo de cabecera que te permite dar modularidad en esta clase de archivo en contexto, se definen nuestras funciones.

En primera instancia, se diseña un algoritmo basado en cómo registrar un usuario, tenemos dos archivos, uno tipo .cpp y otro de tipo. h , en registro_usuario, donde se la interfaz gráfica aplicada fue el de qt creator, primero se diseña el algoritmo o las fases de solución de nuestro problema que sería como hacer un catálogo de productos y que el usuario que es la persona que va a interactuar con el sistema puede elegir unos productos basados en su preferencia y de allí surge la iniciativa de crear un módulo para que el usuario pueda registrarse , allí puede ser nuevo inicio de sesión y donde deben estar los datos validados tienen un valor de conversión en que usa la conversión de todo carácter a minúscula y parámetros de entrada que deben si o si cumplir con las validaciones internas dentro del programa , el primer contacto del usuario y el proyecto es el de registrarse, de allí surge como hacer un algoritmo para que se ingrese un usuario y después se desplieguen todas las demás fases del programa.por consiguiente, la estructura o fases de la solución de este problema se genero los siguientes campos:

- Catalogo.cpp: se crea una lista llamada producto catálogo global = nullptr, aquí viene el catalogo con los productos cargados en sistema, por categoría, tienen las cualidades y descripción de cada producto dentro de una categoría aquí convierte cada descripción de producto a minúsculas y verifica si contiene el texto buscado, verifica y compara el producto o descripción ingresada con la existente en catalogo.
- Catalogo.h: Aquí se declaran varias funciones que están implementadas en otro archivo .cpp. Estas funciones trabajan con listas enlazadas de productos. (ListaProducto) y cadenas de texto (std::string), diseñadas para realizar operaciones específicas. Vamos una por una función por ejemplo: void inicializarCatalogo(); tiene el objetivo de inicializar el catálogo global de productos (catalogoGlobal). cargar datos desde un archivo o establecer valores iniciales que en este caso serían las marcas de productos.
- estructuras.h:para procesar texto de nuestro catálogo de productos y así poder manipular y manejar las cadenas de textos en la consulta del producto de manera más segura la validación de datos de entradas de texto ya que lo convierte internamente a minuscula así hallamos ingresados en mayuscula, hace la comparación y al coincidir se muestra el producto , se maneja mejor con la estructura producto, se define una estructura llamada producto donde tiene los siguientes miembros como: descripción, ID, marca, precio, calidad y categoría.Permite inicializar los atributos del producto (miembros)con valores predeterminados o personalizados. un producto se muestra por su nombre , tipo de producto(descripción), id, del tipo de marca si es nike o no, precio en \$, su calidad (1-5),y categoría de ropa o maquillaje.

- main.cpp: creación del objeto QApplication a(argc, argv):: es necesario para inicializar la aplicación Qt, ya que se encarga de gestionar ventanas, y la interacción con el sistema operativo. recibiendo los argumentos argc y argv para procesar opciones específicas de Qt o de la aplicación, en esta sesión se hace la implementación de todos los archivos funciones y estructuras trabajadas fuera del main pasando del dev c++ al qt creator.
- mainwindow.cpp: desde aquí una ventana y archivo enlazado al qt, mainwindow.h: Declara la clase principal de la ventana (MainWindow), es la interfaz gráfica principal de la aplicación donde ui_mainwindow.h: Es un archivo generado automáticamente por el entorno de desarrollo (como Qt Creator) para el diseño de la interfaz gráfica(botones, menús, etc.) de la ventana principal. La ventana_productos.h: Este archivo define una clase o funciones relacionadas con la ventana que muestra los productos se gestionan y visualizan en otra ventana; por otro lado, hacer registro_usuario.h: la funcionalidad para registrar usuarios donde se incluye formularios de inicio de sesión para ingresar datos como nombre, contraseña, etc. Hay también un usuario.h. Define la clase struct Usuario, que probablemente representa a un usuario en el sistema. Esta clase podría incluir atributos miembro de estructura como nombre, ID, rol, y métodos para gestionar usuarios de acuerdo a las preferencias del producto. El catalogo.h: Este archivo está relacionado con la gestión de catálogo de productos de ropa, zapato y maquillaje. invoca funciones como en el archivo catalogo.h donde se llaman a lista de funciones para inicializar , buscar y obtener elementos (productos)en el catálogo.
- mainwindow.h:Es la declaración de la clase que representa la ventana principal de la aplicación. Esta clase se define en el archivo .h (header) y se implementa en el archivo .cpp. en esta parte se manejan la comunicación del dev c++ con el código y las funciones de la interfaz gráfica donde tiene lugar el llamado por medio de Código para manejar el clic en el botón por ejemplo el de "Ingresar" para abrir sesión de usuario , se trabaja con las señales y de cómo se interactúa el usuario con la interfaz por los click en cada opción y que a su vez se desplieguen ventanas con los requerimientos del proyecto, véase , lista de productos, validación por búsqueda de producto en el catalogo y los registros previos al sistema.
- productodialog.cpp: aquí se inicia una fase de como Inicializa la interfaz gráfica: Crea una instancia de Ui::productodialog y la asigna al puntero ui. Esto permite que el programa acceda y manipule los elementos de la interfaz gráfica diseñados en

Qt así como de botones de inicio. Inicializa la interfaz gráfica de la ventana con setupUi. a partir de aquí se configuran las etiquetas , varias de ellas (lblNombre, lblMarca, lblPrecio, lblCalidad, lblCategoria) para mostrar información de un objeto producto. Establece el título de la ventana como "Detalles del Producto" los detalles son la información (dato) actual del producto si sea precio y características.

- productodialog.h: su definición, que me permita este puntero permite acceder a los elementos de la interfaz gráfica definidos en el archivo .ui correspondiente.
- recomendacion.cpp: aquí es donde se van a presentar unas especies de tablas donde se van a manejar conteoCategorias, conteoMarcas, conteoCalidades: Mapas que almacenan conteos de categorías, marcas y calidades, respectivamente. actualProducto->dato.categoría,actualProducto->dato.marca,actualProducto->dato.calidad: Claves específicas que se extraen del nodo actual. El resultado se incrementan los conteos en los mapas correspondientes para la categoría, marca y calidad del producto actual. el historial de productos,actualHistorial->siguiente: Avanza al siguiente nodo en la lista enlazada del historial. Resultado: El puntero actualHistorial ahora apunta al siguiente nodo, y el proceso puede repetirse para el nuevo dato.
- recomendacion.h: como hacer que incluya el archivo de encabezado estructuras.h, que probablemente contiene definiciones de estructuras, tipos o funciones necesarias para este archivo. Esto permite que el código aquí tenga acceso a esas definiciones. Definición de la estructura Usuario,Una lista (probablemente definida en estructuras.h) que almacena las categorías más frecuentes asociadas al usuario en base a la preferencia.
- registro_usuario.cpp: en la implementación que se pueda cumplir que método para registrar usuario: como: void RegistroUsuario::registrarUsuario() .Defina la lógica para registrar un usuario, incluyendo validaciones y manejo de errores. Mostrar datos (opcional): Se incluye un método adicional mostrarDatos para verificar que los datos se guardaron correctamente. La personalización por si se necesita que esta función haga algo más específico, como validar los datos ingresados (por ejemplo, verificar que el password tenga un formato válido o que la), puedes agregar lógica adicional dentro del método accept().

- registro_usuario.h: establecer la relación con interfaces gráficas en proyectos que usan Qt, el namespace Ui suele contener clases generadas automáticamente ,por el compilador de interfaces gráficas (como uic en Qt). Estas clases representan las interfaces de usuario diseñadas en archivos .ui.
- usuario.cpp: Gestión de usuarios para usar la funcionalidad de usuario.h para autenticar o registrar usuarios.Utilidades: Usar funciones genéricas de utilidades.h para tareas comunes, como formatear datos o validar entradas. Ver catálogo y usar catalogo.h para mostrar, buscar o gestionar elementos en un catálogo, la inclusión del uso de librerías estándar:std::cout y std::cin para interactuar con el usuario.
- usuario.h: se hace un preámbulo de #pragma once que es una directiva de preprocessador que evita que el archivo de cabecera se incluya más de una vez en el mismo archivo fuente. Esto previene errores de redefinición y mejora la eficiencia de compilación el #include "estructuras.h" que Incluya un archivo de cabecera llamado estructuras.h, que probablemente contiene definiciones de estructuras como Nodoarbol y Usuario. estructuras esenciales para el funcionamiento del programa., para el declarar las funciones, la declaración de funciones consista que el programa declara varias funciones que trabajan con un árbol binario (probablemente un árbol de búsqueda binaria) para gestionar usuarios.
- utilidades.cpp: aqui se dan los procesos de validacion ,verificar si hay un salto de línea pendiente y lo descarta,lee una línea completa desde la entrada estándar. Identifica los límites de los caracteres significativos (no espacios) en la cadena. Si la cadena está vacía o contiene solo espacios, devuelve una cadena vacía. Si no, recorta los espacios al principio y al final y devuelve la cadena resultante.
- utilidades.h: esta funciones en este archivo, probablemente genera un identificador único (ID) en forma de cadena de texto. para asignar identificadores a usuarios. Puede leer una entrada de texto desde el usuario. asi como de generar una referencia a la lista de productos que se desea.
- ventana_productos.cpp:Este archivo nos lleve a la implementacion de punteros,que permitan acceder y manipular los elementos visuales de la ventana, como botones, etiquetas, listas, etc. Asi como de diversos eventos durante el programa.
- ventana_producto.h: asi como acceder a otros archivos en ese ala llamado como por ejemplo: que aquí se incluye otro archivo de encabezado llamado usuario.h.

Este archivo probablemente contiene la definición de una clase o funciones relacionadas con un "usuario". El uso de comillas ("usuario.h") indica que es un archivo de encabezado personalizado después de la fase de registro.

Estructuras de Datos Utilizados

- Struct
- Árbol binario
- Lista Circular
- Nodos

Tabla de funciones asociadas a las estructuras

mainwindow (Gestión de Ventana Principal)	<ul style="list-style-type: none">• on_Regis_clicked(): Abre ventana de registro.• on_Inicio_clicked(): Muestra formulario de login.• on_ingresar_clicked(): Valida credenciales de usuario.• on_salir_clicked(): Cierra la aplicación.• on_volver_clicked(): Regresa a pantalla inicial.
registro_usuario (Registro de Nuevos Usuarios)	<ul style="list-style-type: none">• on_buttonBox_accepted(): Valida y registra nuevo usuario.• on_buttonBox_rejected(): Cancela registro.
	<ul style="list-style-type: none">• llenarFiltros(): Carga categorías/marcas en comboboxes.• cargarProductosPorCategoria(): Muestra productos por categoría.

ventana_productos (Interfaz Principal de Productos)	<ul style="list-style-type: none"> • cargarRecomendaciones(): Genera y muestra recomendaciones. • mostrarDetallesProducto(): Abre diálogo con detalles de producto. • on_pushButton_clicked(): Ejecuta búsqueda con filtros. • on_Volver_clicked(): Regresa a pantalla principal.
recomendacion (Sistema de Recomendaciones)	<ul style="list-style-type: none"> • generarRecomendaciones(): Coordina proceso completo. • obtenerEstadisticasUsuario(): Calcula preferencias de usuario. • actualizarPreferenciasDinamicas(): Ajusta preferencias basado en historial. • obtenerTopN(): Selecciona N elementos más frecuentes. • ordenarPorRelevancia(): Ordena productos por relevancia.
catalogo (Gestión de Productos)	<ul style="list-style-type: none"> • inicializarCatalogo(): Carga datos iniciales de productos. • buscarPorCategoria(): Filtra productos por categoría. • buscarPorRangoPrecios(): Filtra por rango de precios. • buscarPorMarca(): Filtra por marca. • buscarPorDescripcion(): Búsqueda textual en descripciones. • obtenerCategoriasUnicas(): Lista categorías disponibles. • obtenerMarcasUnicas(): Lista marcas disponibles.
usuario (Gestión de Usuarios)	<ul style="list-style-type: none"> • crearNodo(): Inicializa nodo de árbol. • insertar(): Inserta usuario en árbol binario. • buscar(): Busca usuario en árbol. • verificarPassword(): Compara contraseñas.

utilidades (Funciones Auxiliares)	<ul style="list-style-type: none"> ● generarID(): Crea identificador único. ● aMinusculas(): Normaliza texto a minúsculas. ● insertarLista(): Agrega elemento a lista enlazada. ● existeEnLista(): Verifica existencia en lista. ● insertarEnLista(): Agrega producto a lista. ● longitudLista(): Calcula tamaño de lista. ● barajarLista(): Aleatoriza orden de lista. ● buscarEnSet(): Verifica existencia en conjunto. ● insertarEnSet(): Agrega elemento a conjunto. ● incrementarEnMapa(): Aumenta contador en mapa. ● mapaEstaVacio(): Verifica si mapa está vacío.
productodialog (Diálogo de Detalles de Producto)	on_pushButton_clicked(): Cierra el diálogo

Análisis de cómo implemento los algoritmos de recomendación por marca

Primera parte, se va a desglosar el código en base al archivo de implementación como es el de recomendación.cpp:

*Estas son bibliotecas estándar de C++ que proporcionan funcionalidades predefinidas.

Por qué están entre < >: Esto indica que son parte de la biblioteca estándar de C++ y no archivos locales.

Propósito de cada una:

<algorithm>: Contiene funciones para manipular datos, como ordenar (std::sort), buscar (std::find), transformar, etc.

<vector>: Proporciona la clase std::vector, que es un contenedor dinámico similar a un arreglo, pero con tamaño ajustable.

<unordered_map>: Define un contenedor tipo tabla hash (std::unordered_map) que permite almacenar pares clave-valor con acceso rápido

Usa archivos personalizados para organizar su lógica en módulos (recomendaciones, utilidades, catálogo).

Está diseñado para ser modular y reutilizable, separando las funcionalidades en diferentes archivos .cpp.funcion .h y main.cpp

es proyecto bien estructurado, ya que mejora la legibilidad, el mantenimiento y la escalabilidad del código.

*/

```
ListaString obtenerTopN(MapaStringInt mapa, int n) { //Función: ListaString  
obtenerTopN(MapaStringInt mapa, int n)
```

//función parece estar diseñada para obtener los "top N" elementos de un mapa
(probablemente un mapa personalizado)

/*declaracion de la funcion:

ListaString: Es el tipo de dato que devuelve la función. Parece ser una lista personalizada que almacena cadenas de texto (string).

MapaStringInt mapa: Es un parámetro de entrada, que parece ser un mapa personalizado que asocia cadenas (string) con enteros (int).

int n: Es otro parámetro de entrada que indica cuántos elementos (los "top N") se desean obtener.

```
*/  
ListaString resultado = nullptr;  
int longitud = 0;  
NodoMapaStringInt* actual = mapa;  
while(actual != nullptr) {  
    /*bucle while para recorrer el mapa:
```

Condición (actual != nullptr): El bucle se ejecuta mientras el puntero actual no sea nullptr, es decir, mientras haya nodos en el mapa.

longitud++: En cada iteración, se incrementa la variable longitud para contar cuántos nodos hay en el mapa.

actual = actual->siguiente: Se avanza al siguiente nodo del mapa utilizando el puntero siguiente. apunta siguiente

```
*/  
  
longitud++; // acumula incremento  
actual = actual->siguiente;  
/*variables locales:  
ListaString resultado = nullptr;  
int longitud = 0;  
NodoMapaStringInt* actual = mapa;
```

resultado: Inicialmente se define como nullptr. Probablemente se usará para almacenar la lista de resultados que se devolverá al final.

longitud: Se inicializa en 0 y se usará para contar la cantidad de nodos en el mapa.

actual: Es un puntero que apunta al inicio del mapa (mapa). Se usará para recorrer los nodos del mapa

la función tiene como objetivo contar la cantidad de nodos en el mapa (longitud). para determinar que seleccionar en el mapa Los almacene en la lista resultado.

Devuelva la lista resultado del producto.(la seleccion de elementos asociado a texto ingresado

*/

}

for (int i = 0; i < n && i < longitud; ++i) {//algoritmo que selecciona los mejores elementos de un mapa enlazado basado en ciertos criterios.

/*Propósito: Este bucle itera hasta un máximo de n veces o hasta que se alcance la longitud especificada (longitud).

Variables: i: Contador que comienza en 0 y se incrementa en cada iteración y Condición: El bucle continúa mientras i sea menor que n y menor que longitud.

*/

NodoMapaStringInt* mejor = nullptr;

NodoMapaStringInt* ptr = mapa;

/* declaracion de punteros:

mejor: Se inicializa como nullptr y se usará para almacenar el nodo con el mejor valor encontrado en esta iteración.

ptr: Apunta al inicio de la estructura mapa, que parece ser una lista enlazada de nodos.

*/

while (ptr != nullptr) {// bucle while interno

/*Propósito: Recorre todos los nodos de la lista enlazada mapa.

Condición: El bucle continúa mientras ptr no sea nullptr, es decir, mientras no se haya llegado al final de la lista.

*/

```
/*verificacion de existencia del elemento en la lista:
```

Propósito: Comprueba si la clave del nodo actual (`ptr->clave`) ya está en la lista resultado.

Función `existeEnLista`: Se asume que esta función devuelve true si la clave ya está en resultado, y false en caso contrario.

```
*/
```

```
if (!existeEnLista(resultado, ptr->clave)) {
```

```
    if (mejor == nullptr || ptr->valor > mejor->valor) {
```

```
        mejor = ptr;
```

```
        /*seleccion del mejor nodo :
```

Si mejor aún no ha sido asignado (`nullptr`), se asigna el nodo actual (`ptr`).

Si el valor del nodo actual (`ptr->valor`) es mayor que el valor del nodo almacenado en `mejor`,

se actualiza `mejor` para que apunte al nodo actual.(encontrar el nodo con el valor mas alto)

```
    */
```

```
}
```

```
}
```

`ptr = ptr->siguiente;`//Propósito: Mueve el puntero `ptr` al siguiente nodo en la lista enlazada

```
}
```

```
if (mejor != nullptr) {
```

```
    insertarLista(resultado, mejor->clave);
```

```
    /*insercion en la lista resultado:
```

Condición: Si se encontró un nodo válido (`mejor` no es `nullptr`), se inserta su clave en la lista resultado.

Función `insertarLista`: Se asume que esta función agrega la clave a la lista resultado.

```
*/
```

```
}
```

```
}

return resultado;//retorno de resultado:

//Propósito: Devuelve la lista resultado, que contiene las claves de los mejores nodos
seleccionados.
```

/*/////////////////por lo tanto en este segmento de codigo:

El bucle principal controla cuántos elementos se seleccionan (máximo n o longitud).

En cada iteración, se recorre la lista enlazada mapa para encontrar el nodo con el valor más alto que aún no esté en resultado.

Si se encuentra un nodo válido, su clave se agrega a resultado ,al final, se devuelve la lista resultado.

*/

///////////////estructuras de datos involucradas:

/* Estructura de datos:

NodoMapaStringInt una estructura que representa un nodo de una lista enlazada, con al menos los siguientes campos:

clave: La clave del nodo.

valor: El valor asociado a la clave.

siguiente: Un puntero al siguiente nodo en la lista.

///////////////funciones auxiliares en la lista

existeEnLista: Comprueba si una clave ya está en la lista resultado.

insertarLista: Inserta una clave en la lista resultado.*/

}

EstadisticasUsuario obtenerEstadisticasUsuario(Usuario *usuario)

```
{  
    EstadisticasUsuario stats;  
    MapaStringInt conteoCategorias = nullptr;  
    MapaStringInt conteoMarcas = nullptr;  
    MapaIntInt conteoCalidades = nullptr;  
    /*el uso de nullptr; inicializar las variables con nullptr indica que estas aún no están  
    asociadas a un objeto válido en memoria.  
*/
```

/*

definicion de la funcion :

EstadisticasUsuario obtenerEstadisticasUsuario(Usuario *usuario):

Es una función que devuelve un objeto de tipo EstadisticasUsuario.

Recibe como parámetro un puntero a un objeto de tipo Usuario (Usuario *usuario).

El propósito de esta función parece ser calcular o recopilar estadísticas relacionadas con el usuario proporcionado.

definicion de las variables locales:

Dentro de la función, se declaran varias variables locales que probablemente se usarán para procesar y almacenar datos relacionados

con las estadísticas del usuario:

EstadisticasUsuario stats:

Es una instancia de la clase o estructura EstadisticasUsuario. Se usará para almacenar las estadísticas calculadas y será el valor que se devuelva al final de la función.

MapaStringInt conteoCategorias:

Es una variable que parece ser un mapa (o diccionario) que relaciona cadenas de texto (string) con enteros (int). Se inicializa con nullptr, lo que indica que aún no apunta a un objeto válido.

MapaStringInt conteoMarcas:

Similar a conteoCategorias, es otro mapa que relaciona cadenas de texto con enteros, también inicializado como nullptr.

MapaIntInt conteoCalidades:

Es un mapa que relaciona enteros con enteros, también inicializado como nullptr

*/

```
Nodolista *actualHistorial = usuario->historial;
```

```
while (actualHistorial != nullptr)
```

```
{
```

```
    int id = std::stoi(actualHistorial->dato);
```

```
    NodoProducto* actualProducto = catalogoGlobal;
```

```
    while (actualProducto != nullptr)
```

```
    {// BUCLE PARA RECORRER LA LISTA DE LOS PRODUCTOS:
```

```
        /*Este bucle while recorre la lista enlazada de productos (catalogoGlobal) nodo por nodo.
```

Condición: El bucle continúa mientras actualProducto no sea nullptr, lo que indica que aún hay nodos por procesar.

*/

```
if (actualProducto->dato.id == id)
```

```
{/*COMPARACION DE IDS
```

```
    actualProducto->dato.id: Accede al atributo id del dato almacenado en el nodo actual de la lista de productos.
```

id: Es el entero obtenido previamente de actualHistorial->dato.

Condición: Si el id del producto actual coincide con el id del historial, se ejecuta el bloque de código dentro del if.

```
*/
```

```
incrementarEnMapa(conteoCategorias, actualProducto->dato.categoría);  
incrementarEnMapa(conteoMarcas, actualProducto->dato.marca);  
incrementarEnMapa(conteoCalidades, actualProducto->dato.calidad);  
break;  
/* conteo de mapas de listas
```

incrementarEnMapa: Parece ser una función personalizada que actualiza un mapa (probablemente un std::map o std::unordered_map) incrementando el valor asociado a una clave.

Parámetros:

conteoCategorias, conteoMarcas, conteoCalidades: Mapas que almacenan conteos de categorías, marcas y calidades, respectivamente.

actualProducto->dato.categoría, actualProducto->dato.marca, actualProducto->dato.calidad: Claves específicas que se extraen del nodo actual.

Resultado: Se incrementan los conteos en los mapas correspondientes para la categoría, marca y calidad del producto actual.

con break: hace la salida del bucle

Si se encuentra un producto cuyo id coincide con el del historial, se ejecuta el break para salir del bucle.

Esto implica que no es necesario seguir buscando en la lista de productos.

```
*/
```

```
}
```

```
actualProducto = actualProducto->siguiente;
```

//actualHistorial->siguiente: Avanza al siguiente nodo en la lista enlazada del historial.

//Resultado: El puntero actualHistorial ahora apunta al siguiente nodo, y el proceso puede repetirse para el nuevo dato.

}

actualHistorial = actualHistorial->siguiente;

/*

///////////////AQUI ESTA LA CONVERSION DE DATO A ENTERO:

int id = std::stoi(actualHistorial->dato);

std::stoi: Convierte una cadena de texto (std::string) en un número entero.

actualHistorial->dato: Se asume que actualHistorial es un puntero a un nodo de una lista enlazada, y dato es un atributo que contiene un valor en formato de texto que representa un número.

Resultado: El valor de dato se convierte a un entero y se almacena en la variable id.

Inicialización de un puntero para recorrer otra lista

NodoProducto*: Declara un puntero a un nodo de tipo NodoProducto.

catalogoGlobal: Se asume que es el puntero al inicio de otra lista enlazada que contiene productos.

Resultado: actualProducto comienza apuntando al primer nodo de la lista catalogoGlobal.

/////////////por lo tanto esta parte del codigo Nodolista *actualHistorial = usuario->historial; tiene:

Se obtiene un id del nodo actual de la lista actualHistorial.

Se recorre la lista catalogoGlobal buscando un producto cuyo id coincida con el del historial.

Si se encuentra una coincidencia:

Se actualizan los mapas de conteo para categoría, marca y calidad.

Se sale del bucle.

Se avanza al siguiente nodo en la lista del historial.

*/

}

```
if (!mapaEstaVacio(conteoCategorias))
{
    stats.categoriasFrecuentes = obtenerTopN(conteoCategorias, 2);
    /*
}
```

la condicion if evalua:

Aquí se está evaluando si el mapa (o contenedor) llamado conteoCategorias no está vacío.

La función mapaEstaVacio(conteoCategorias) probablemente verifica si el contenedor conteoCategorias tiene elementos.

El operador lógico ! (negación) invierte el resultado de la función. Es decir:

Si mapaEstaVacio(conteoCategorias) devuelve true (el mapa está vacío), entonces !true será false y el bloque de código dentro del if no se ejecutará.

Si mapaEstaVacio(conteoCategorias) devuelve false (el mapa no está vacío), entonces !false será true y el bloque de código se ejecutará.

para la parte de stas.categoriasFrecuentes=obtenerTopN(conteoCategorias,2);

Si el mapa conteoCategorias no está vacío, se ejecuta esta línea.

Aquí se está asignando un valor a la propiedad categoriasFrecuentes del objeto stats.

La función obtenerTopN(conteoCategorias, 2) probablemente toma el mapa conteoCategorias como entrada y devuelve los 2 elementos más frecuentes o relevantes del mapa.

El resultado de obtenerTopN se almacena en stats.categoriasFrecuentes.

/////////////////////////////por lo tanto, evalua

Este fragmento de código verifica si el contenedor conteoCategorias tiene datos. Si tiene, calcula los 2 elementos más destacados (o frecuentes) y los guarda en stats.categoriasFrecuentes.

así como: conteoCategorias es un mapa que contiene categorías y su frecuencia y Si el mapa no está vacío, obtenerTopN(conteoCategorias, 2) podría devolver

// dos categorías mas frecuentes y este mismo resultado se guarda en resultado se asignaría a stats.categoriasFrecuentes.

```
*/  
}  
  
if (!mapaEstaVacio(conteoMarcas))  
{  
    stats.marcasFrecuentes = obtenerTopN(conteoMarcas, 2);
```

/*condicion del if:

Condición del if:

La condición !mapaEstaVacio(conteoMarcas) verifica si el mapa (o estructura de datos) llamado conteoMarcas no está vacío.

mapaEstaVacio(conteoMarcas) es probablemente una función que devuelve true si el mapa está vacío y false si contiene elementos.

El operador lógico ! (negación) invierte el resultado de la función. Por lo tanto:

Si mapaEstaVacio(conteoMarcas) devuelve false (el mapa tiene datos), la condición del if será verdadera.

Si el mapa está vacío, la condición será falsa y el bloque de código dentro del if no se ejecutará.

estructura del if:

Si el mapa no está vacío, se ejecuta el bloque de código dentro de las llaves {}.

Dentro de este bloque, se asigna un valor a stats.marcasFrecuentes. Esto probablemente sea un atributo de un objeto llamado stats.

Llamada de la función para obtener top

La función obtenerTopN(conteoMarcas, 2) parece tomar dos argumentos:

conteoMarcas: El mapa o estructura de datos que contiene las marcas y sus conteos.

2: Un número entero que indica que se deben obtener los 2 elementos más frecuentes o relevantes del mapa.

El resultado de esta función se asigna a stats.marcasFrecuentes.

////////////// por lo tanto:

verifica si el mapa conteoMarcas contiene datos. Si no está vacío, utiliza la función obtenerTopN

para extraer los 2 elementos más relevantes (por ejemplo, las marcas más frecuentes)

Ahora bien despues de copiar codigo, nos centramos en el desplazamiento por historial de interacción es decir al elegir una preferencia se despliega las demas productos similares y de coincidencia:

```
Nodolista *actualHistorial = usuario->historial;  
while (actualHistorial != nullptr)  
{  
    int id = std::stoi(actualHistorial->dato);//////////  
    NodoProducto* actualProducto = catalogoGlobal;  
    while (actualProducto != nullptr)  
    {  
        if (actualProducto->dato.id == id)  
        {  
            incrementarEnMapa(conteoCategorias, actualProducto->dato.categoría);  
            incrementarEnMapa(conteoMarcas, actualProducto->dato.marca);  
            /*Propósito: Una vez que se encuentra el producto con el id correspondiente,  
             no es necesario seguir recorriendo la lista, por lo que se  
             utiliza break para salir del bucle.  
             */  
            break;  
        }  
        actualProducto = actualProducto->siguiente;  
    }  
    /* es el avance al siguiente nodo de la lista  
     actualProducto = actualProducto->siguiente;  
actualProducto->siguiente: Apunta al siguiente nodo en la lista enlazada de productos.  
Resultado: El puntero actualProducto avanza al siguiente nodo.*  
*/  
actualHistorial = actualHistorial->siguiente;//////////
```

```
/*Avance al siguiente nodo en la lista de historial
```

Copiar el código

```
actualHistorial = actualHistorial->siguiente;
```

actualHistorial->siguiente: Apunta al siguiente nodo en la lista enlazada de historial.

Resultado: El puntero actualHistorial avanza al siguiente nodo para procesar el siguiente dato.

```
*/
```

```
/* conversion de dato entero:
```

Para la segunda parte , se tiene que para aplicar las recomendaciones se tiene una ventanas de productos bajo el analisis que cada lista insertada un nuevo nodo, producto sucede si, y solo si usuario se registra.

```
connect(ui->listWidget_2, &QListWidget::itemClicked, [this](QListWidgetItem* item)  
{//Primera conexión: connect con una lambda
```

```
    int id = item->data(Qt::UserRole).toInt();
```

```
    mostrarDetallesProducto(id);
```

```
});
```

```
connect(ui->pushButton, &QPushButton::clicked, this,  
&Ventana_Productos::on_pushButton_clicked);
```

```
/* conexión con lambda:
```

connect: Es una función de Qt que conecta una señal (evento) con un slot (función que responde al evento).

ui->listWidget_2: Es un puntero al widget de tipo QListWidget que está en la interfaz gráfica. Aquí se conecta su señal itemClicked.

&QListWidget::itemClicked: Es la señal emitida cuando un elemento de la lista (QListWidgetItem) es clicado.

Lambda [this](QListWidgetItem* item): Es una función anónima (lambda) que se ejecuta cuando la señal itemClicked es emitida.

[this]: Captura el puntero this para acceder a los miembros de la clase actual.

(QListWidgetItem* item): Recibe como argumento el elemento clicado(seleccionado con click)

```
cuerpo de lambda:           int id =  
item->data(Qt::UserRole).toInt();mostrarDetallesProducto(id);
```

item->data(Qt::UserRole): Obtiene un dato asociado al elemento clicado, almacenado bajo el rol Qt::UserRole.

.toInt(): Convierte ese dato a un entero.// hace la conversion pero empleando con qt

mostrarDetallesProducto(id): Llama a una función miembro de la clase actual, pasando el ID del producto para mostrar sus detalles.

2.Segunda conexión: connect con un slot miembro: connect(ui->pushButton, &QPushButton::clicked, this, &Ventana_Productos::on_pushButton_clicked);

solo sucede si es click:

ui->pushButton: Es un puntero al botón de la interfaz gráfica.

&QPushButton::clicked: Es la señal emitida cuando el botón es clicado.

this: Es el puntero a la instancia actual de la clase (probablemente Ventana_Productos).

&Ventana_Productos::on_pushButton_clicked:

Es un puntero a un método miembro de la clase Ventana_Productos. con el click del usuario.

///////////////por lo tanto:

con la primera conexión, usa una lambda para definir el comportamiento directamente en línea.

con la segunda conexión se conecta directamente una señal con un slot miembro predefinido.

aquí se conecta eventos de la interfaz gráfica con funciones que manejan esos eventos tales como:

casos de hacer clic en un elemento de la lista (listWidget_2), se ejecuta una lambda que obtiene un ID y llama a mostrarDetallesProducto.

caso de hacer clic en un botón (pushButton), se ejecuta el método on_pushButton_clicked de la clase Ventana_Productos.

*/

//////////////////////////////7

//realiza ciertas tareas específicas relacionadas con la gestión de datos, como llenar filtros, cargar productos por categoría y cargar recomendaciones.

llenarFiltros();

cargarProductosPorCategoria("Todas las categorías");

cargarRecomendaciones();

}

/*///////////////////funcionalidades del código:

llenarFiltros(); :

Es una llamada a una función llamada llenarFiltros. y

Su propósito parece ser inicializar o llenar los filtros que se usarán en el programa, posiblemente para filtrar productos, categorías u otros datos.

cargarProductosPorCategoria("Todas las categorías");

Es una llamada a una función llamada cargarProductosPorCategoria.

Esta función recibe un parámetro de tipo string (en este caso, "Todas las categorías"), lo que indica que probablemente carga productos relacionados

con una categoría específica. así como la interaccion de lista para obtener y mostrar los productos.

cargarRecomendaciones();

Es una llamada a una función llamada cargarRecomendaciones.

No recibe parámetros, lo que sugiere que genera recomendaciones basadas en datos preexistentes, en base a preferencias del usuario, y productos recomendados

*/

/* por lo tanto , el objetivo tiende a ser como :

Aplicaciones de comercio electrónico: Para mostrar productos y recomendaciones.(algoritmo para negocios)

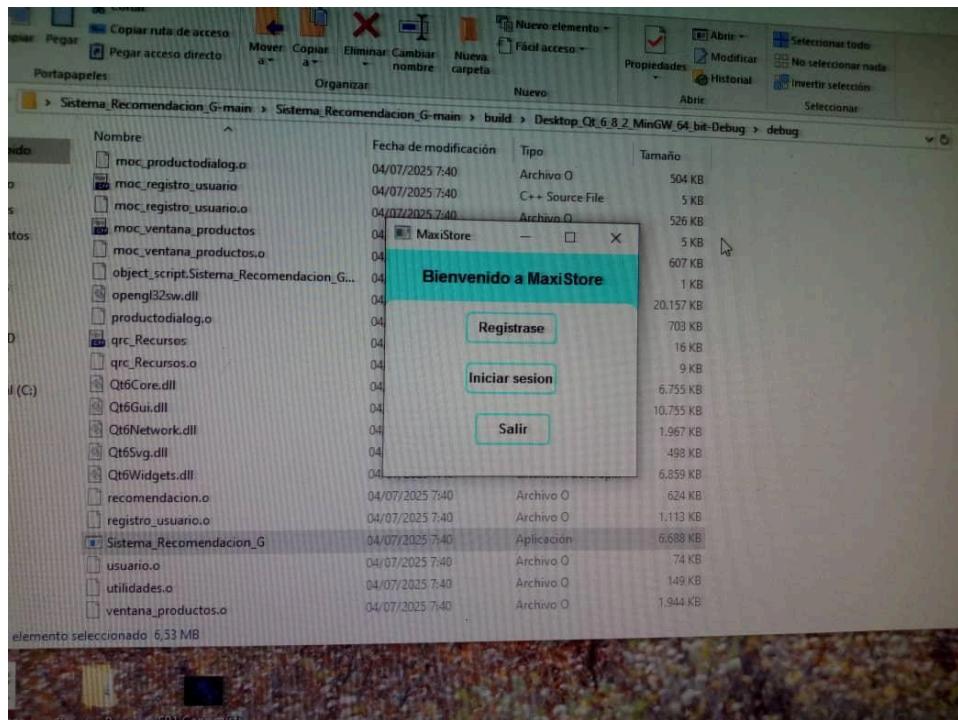
Sistemas de gestión: Para filtrar y cargar datos según categorías.(tipos de marcas)

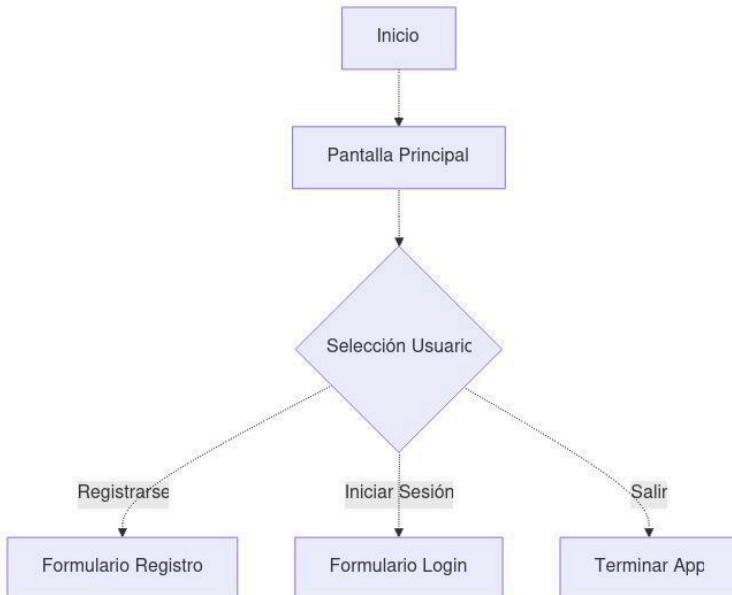
Por lo tanto, struct Nodoarbol define un nodo para un árbol binario, con un dato de tipo Usuario y punteros a los hijos izquierdo y derecho.

struct Recomendaciones organiza listas de productos recomendados según diferentes criterios y permite limitar la cantidad de recomendaciones por sección.

Pantalla de Interfaz Grafica

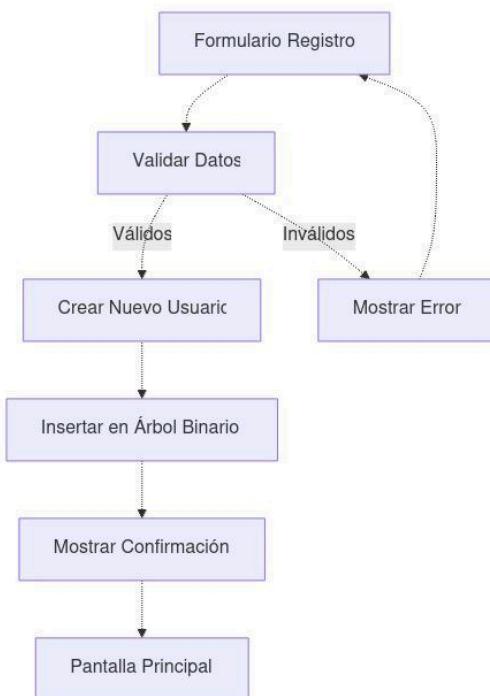
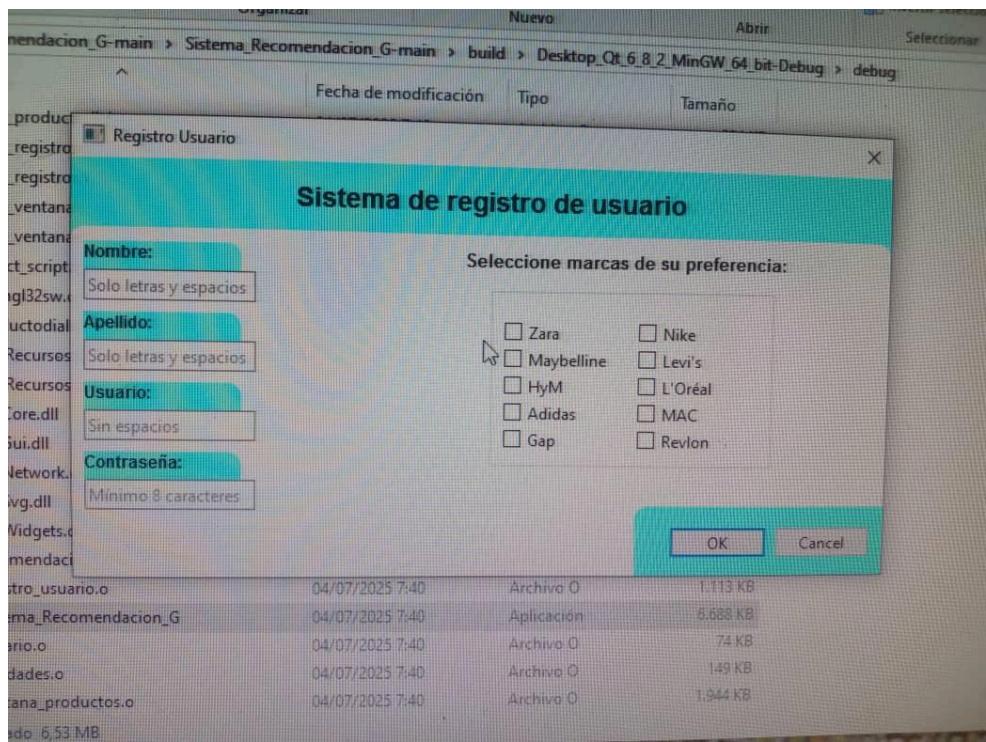
Inicio del programa



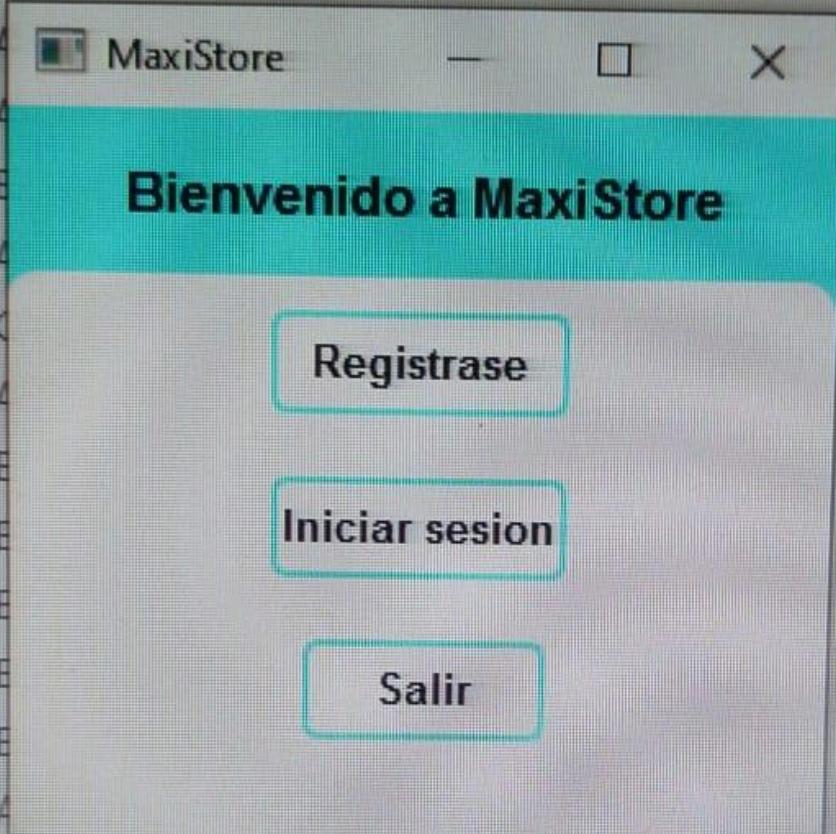


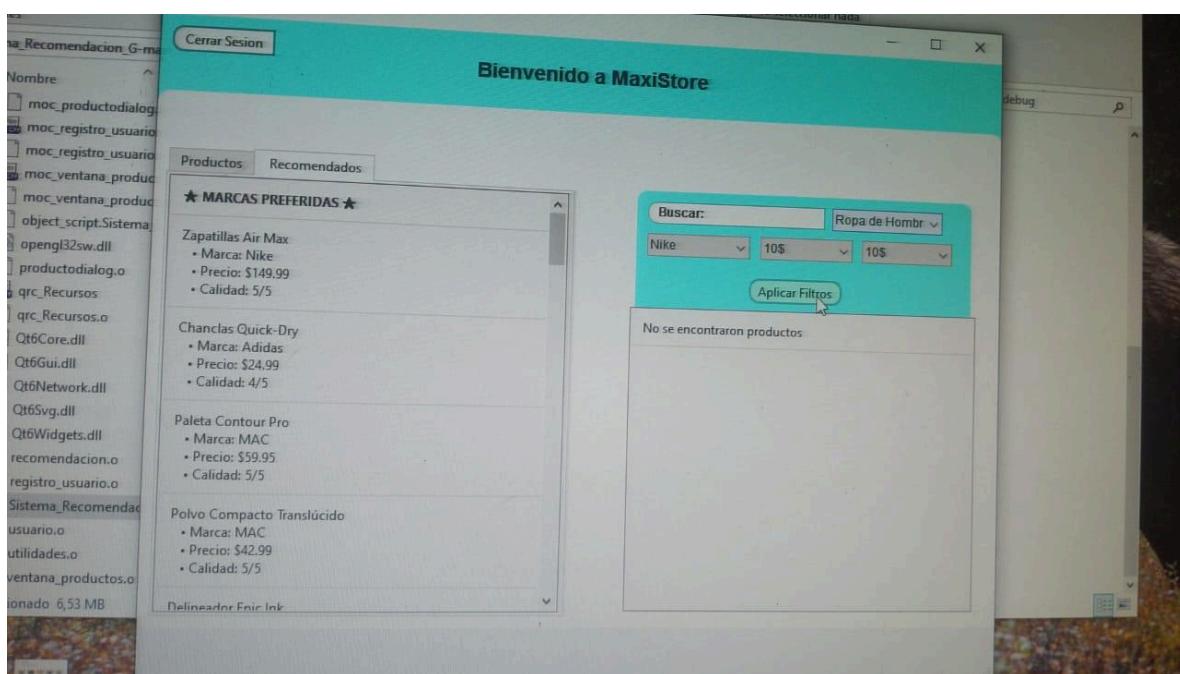
⊕ ⊖ ⊙ ^ v < >

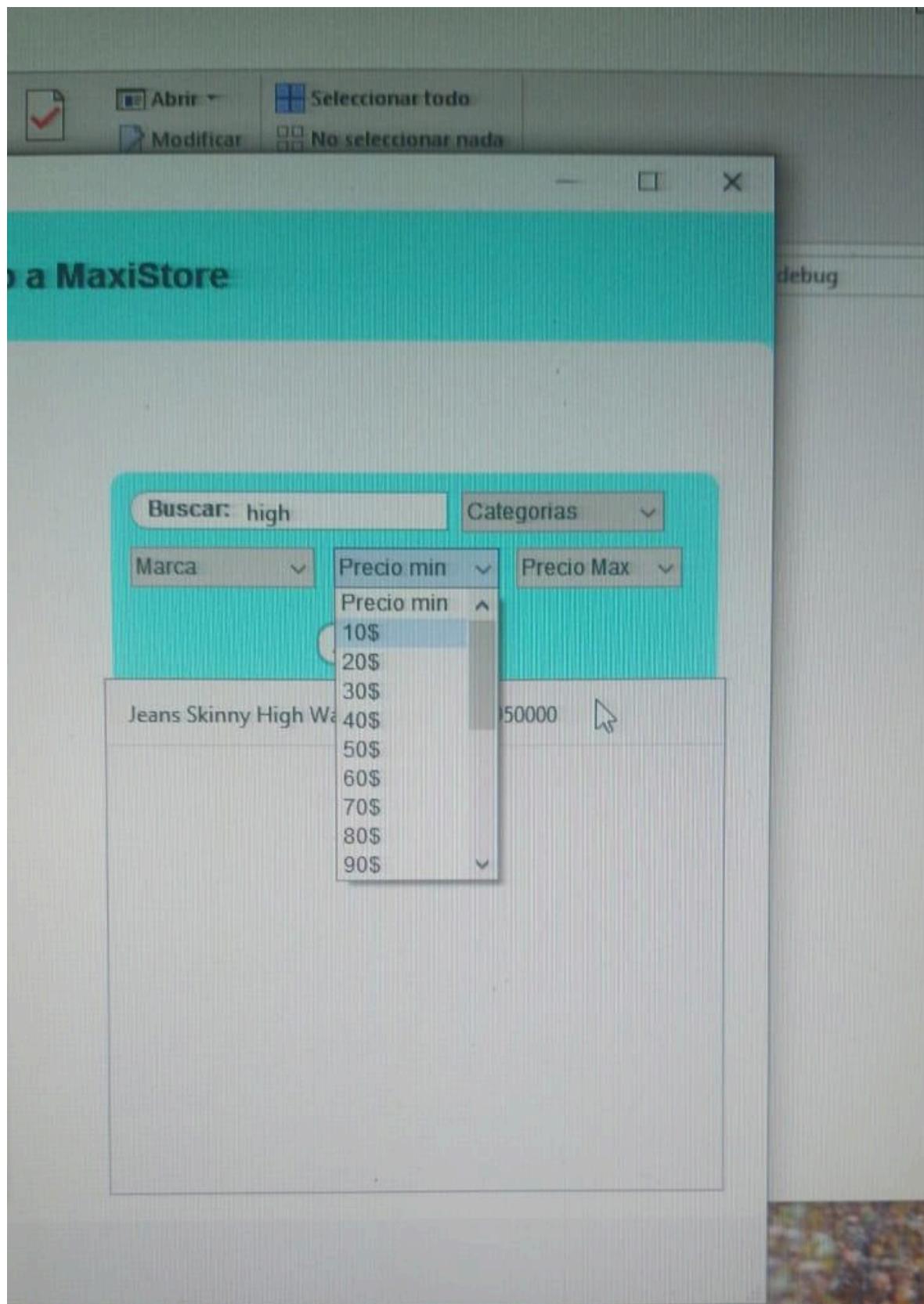
Ventana de Registro de Usuario

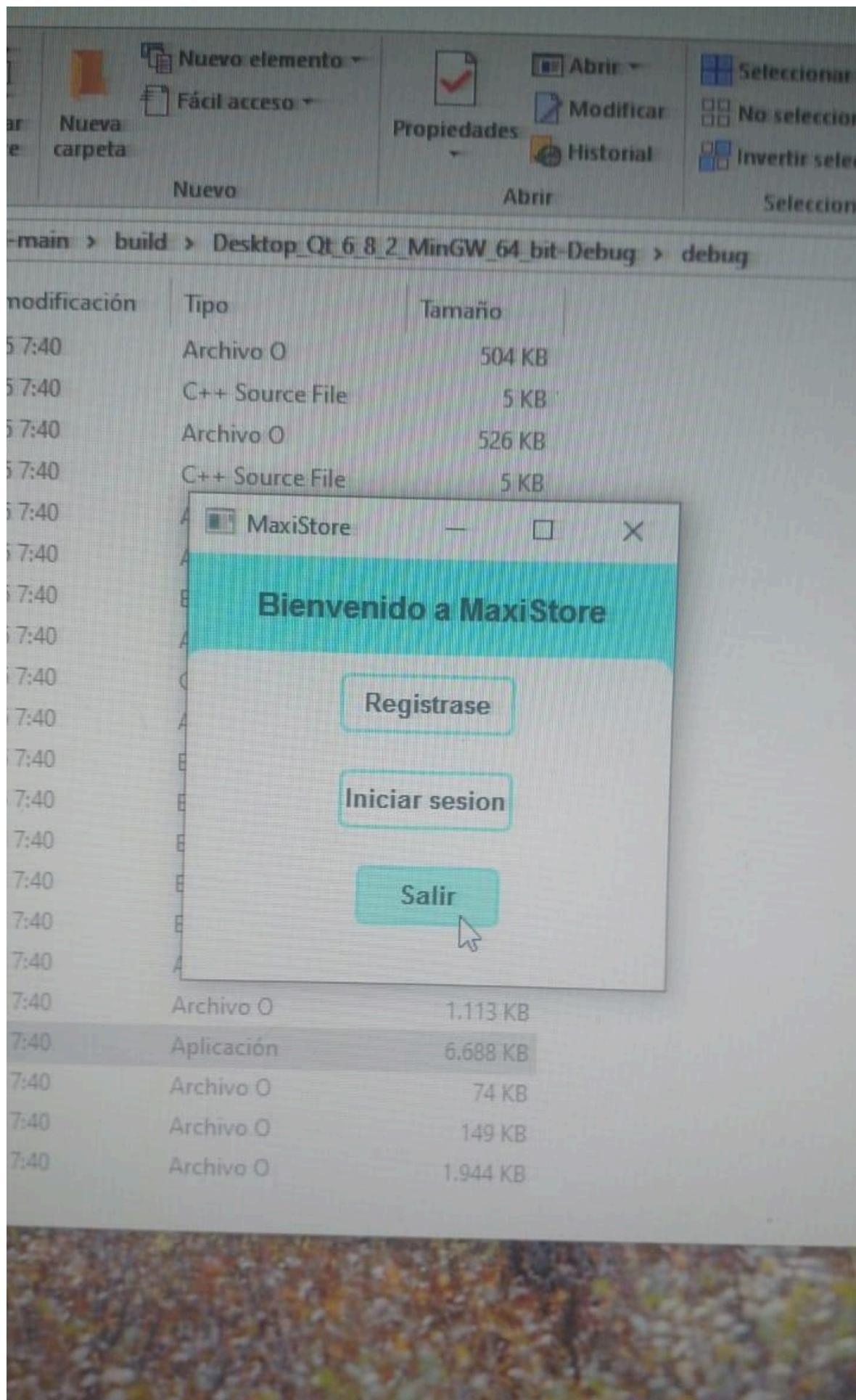


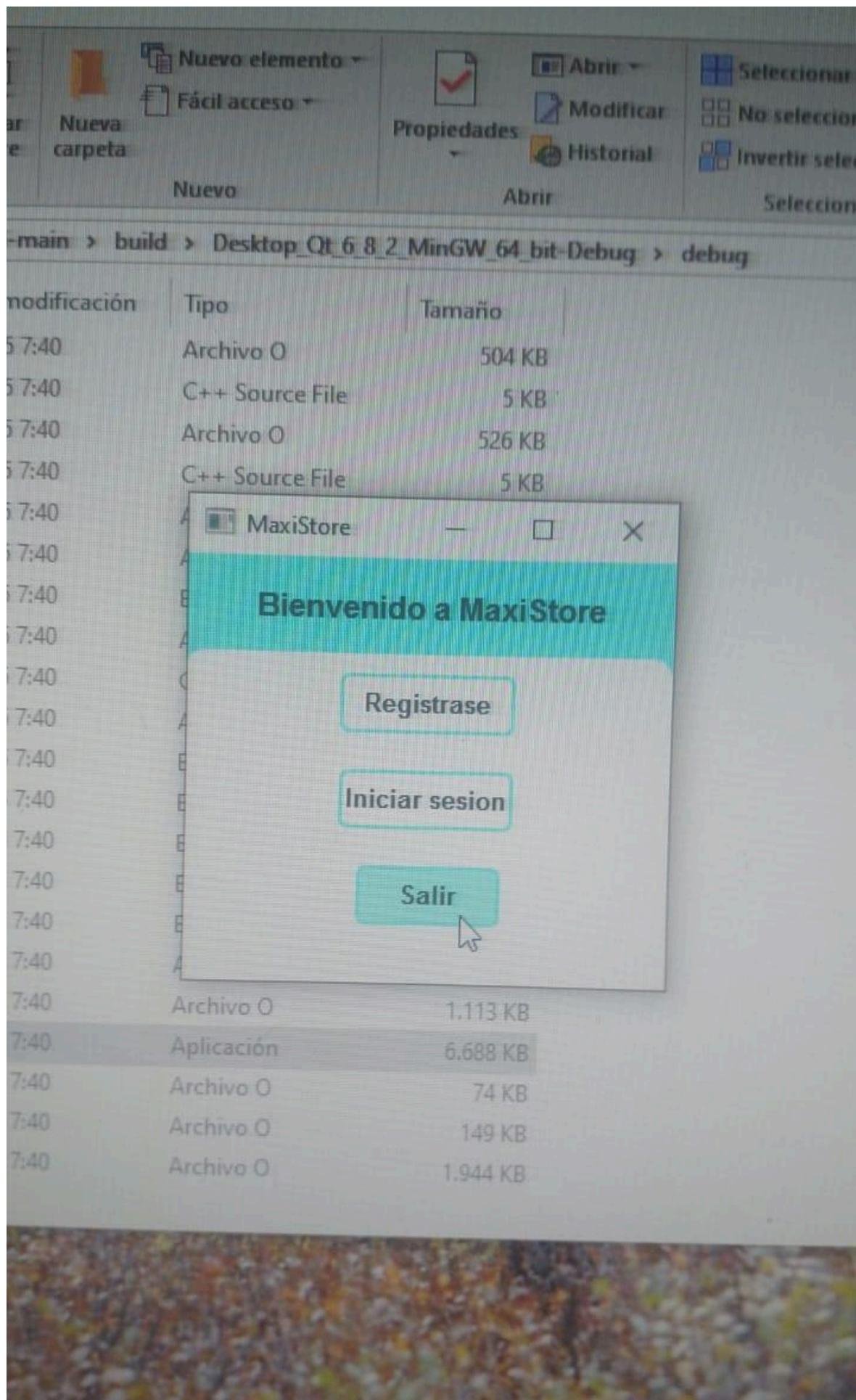
build > Desktop_Qt_6_8_2_MinGW_64_bit-Debug > debug

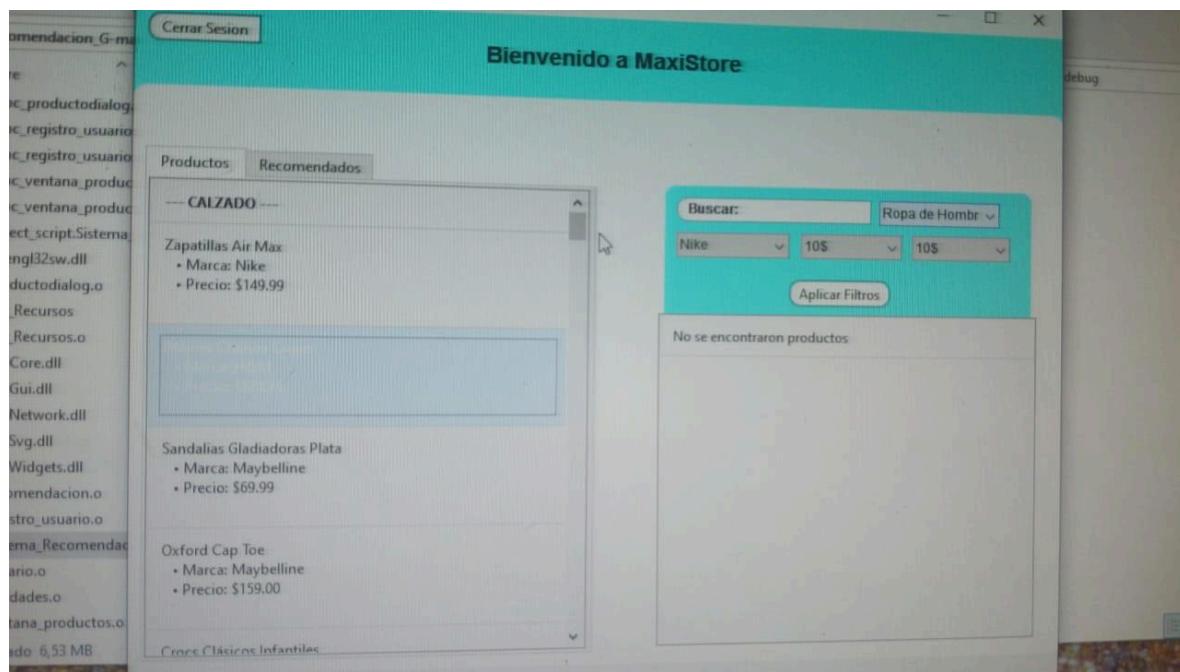
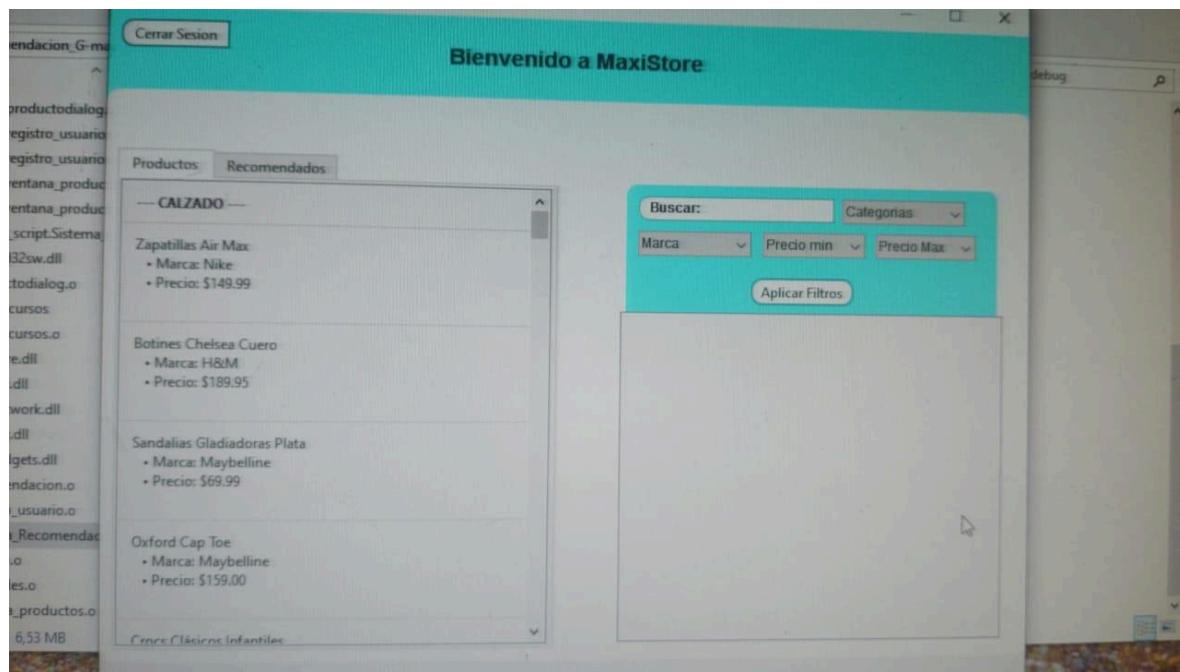
ón	Tipo	Tamaño
	Archivo O	504 KB
	C++ Source File	5 KB
	Archivo O	526 KB
	C++ Source File	5 KB
 A screenshot of a desktop environment showing a file list and a running application window. The application window is titled 'MaxiStore' and has a teal header bar with the text 'Bienvenido a MaxiStore'. It contains three buttons: 'Registrarse', 'Iniciar sesión', and 'Salir'. The window is partially overlapping the file list. <p>MaxiStore</p> <p>Bienvenido a MaxiStore</p> <p>Registrarse</p> <p>Iniciar sesión</p> <p>Salir</p>		
	Archivo O	1.113 KB
	Aplicación	6.683 KB
	Archivo O	74 KB
	Archivo O	149 KB
	Archivo O	1.944 KB

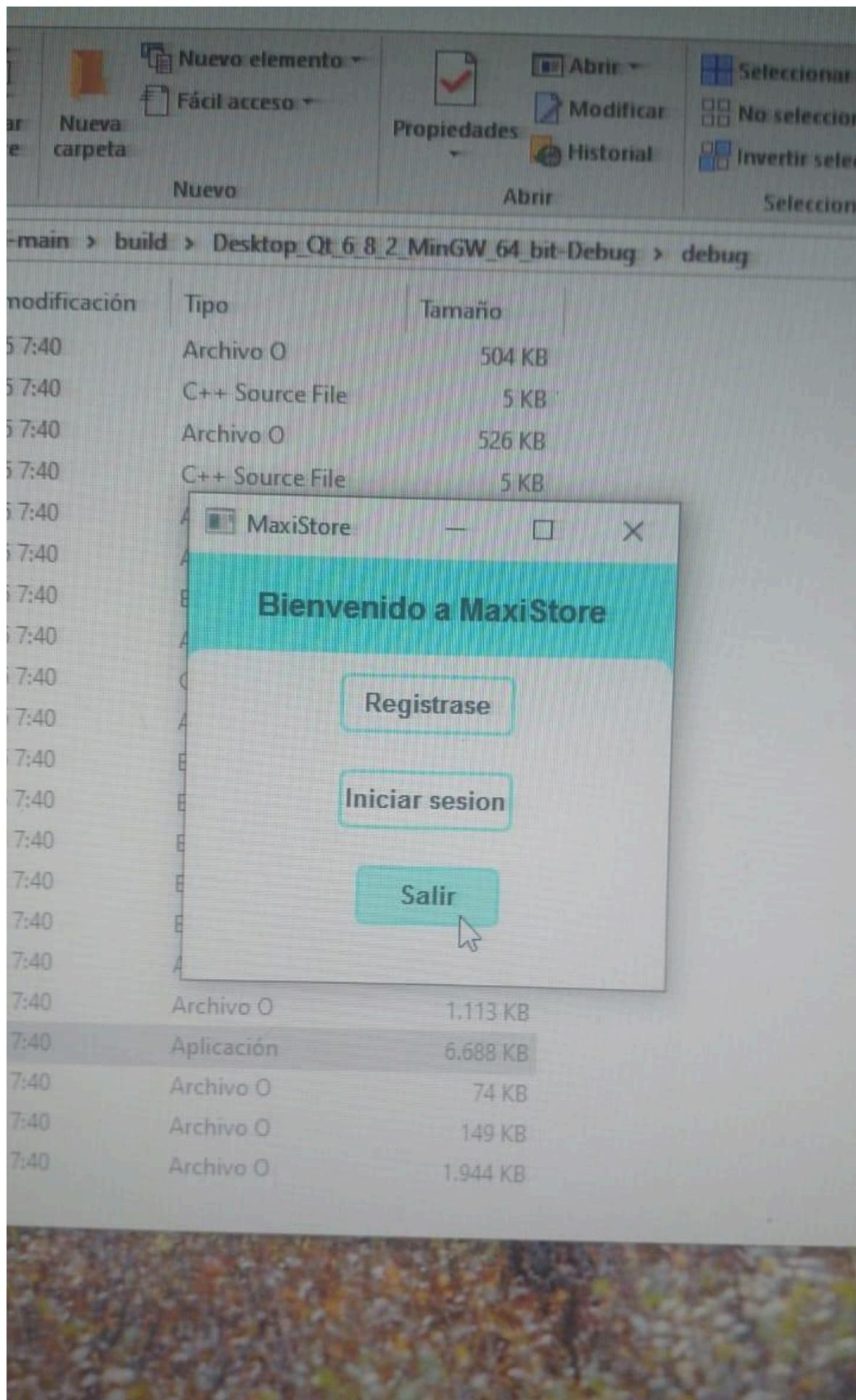


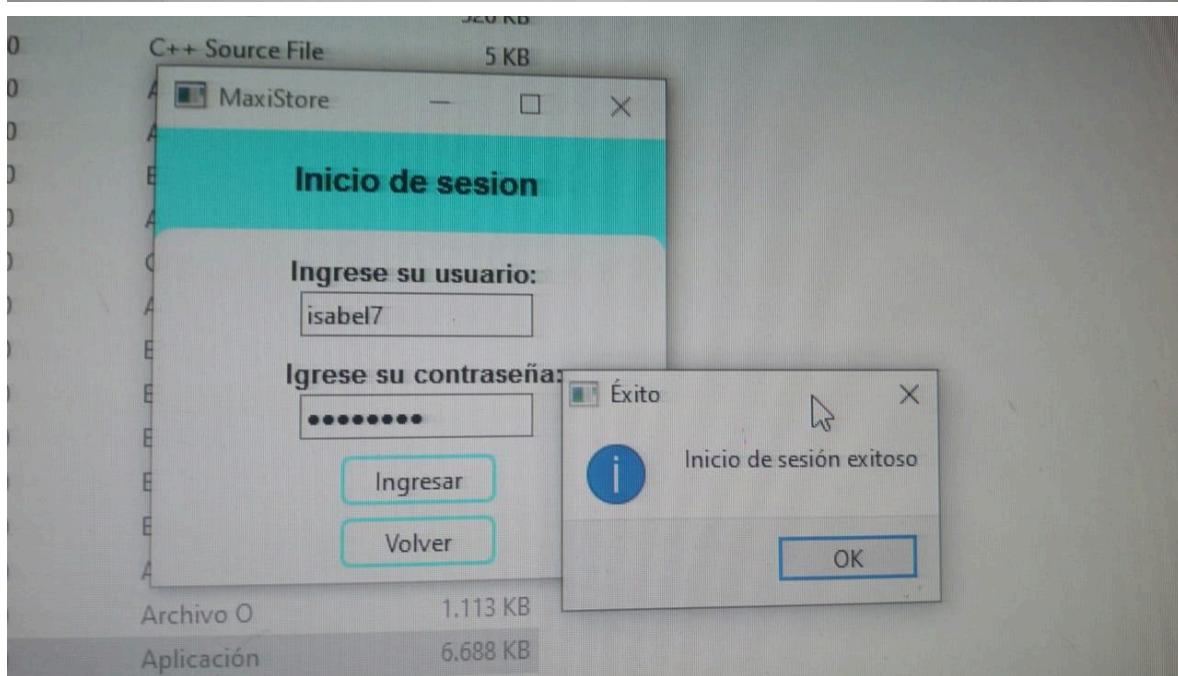
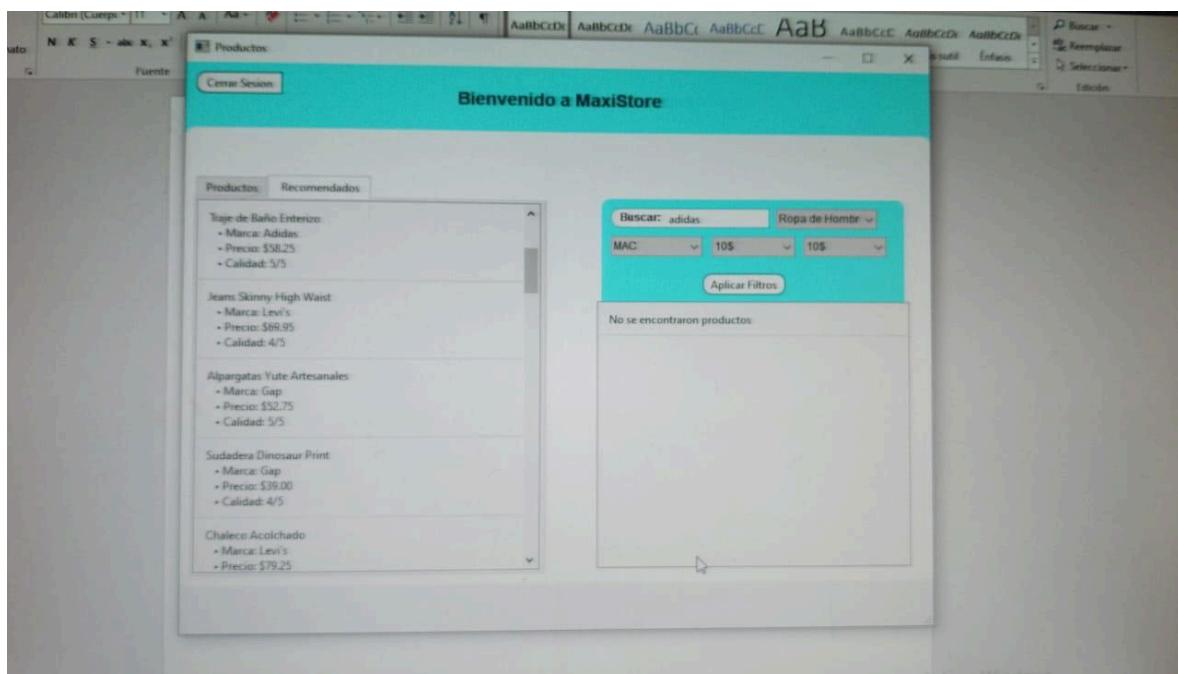


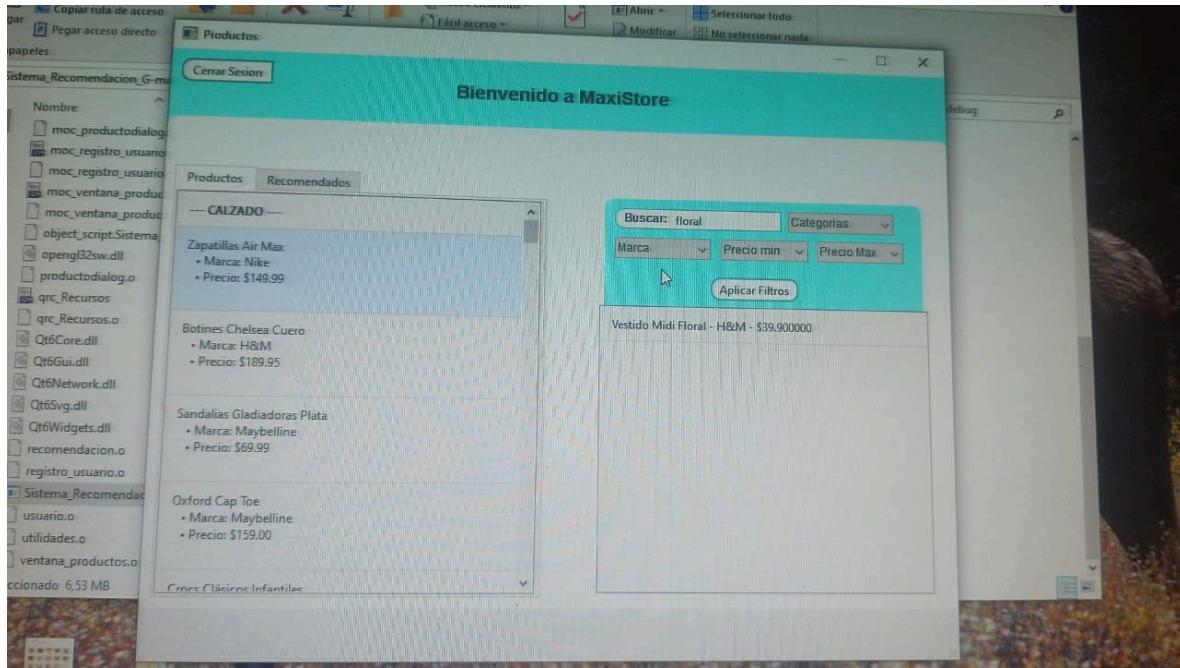


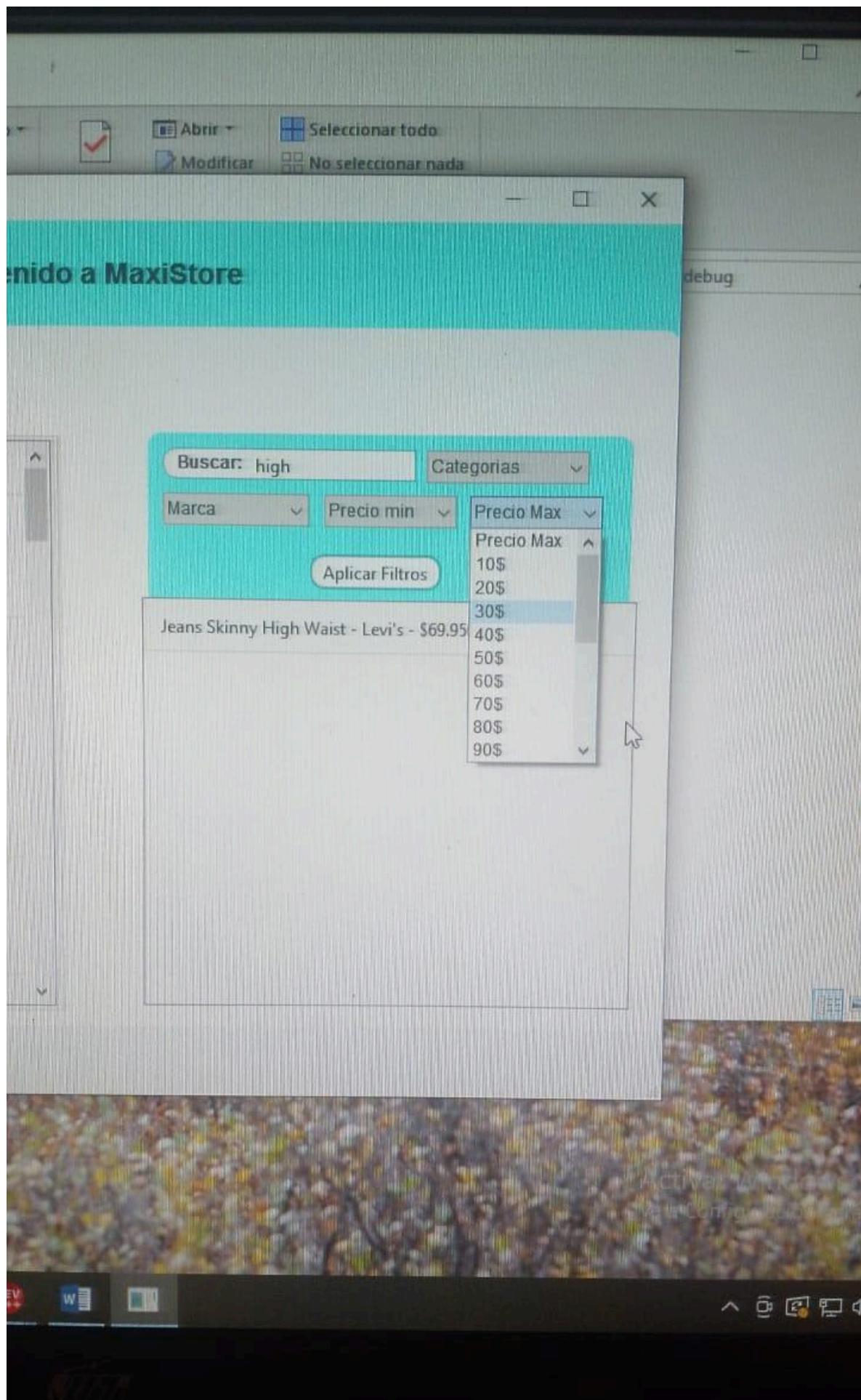


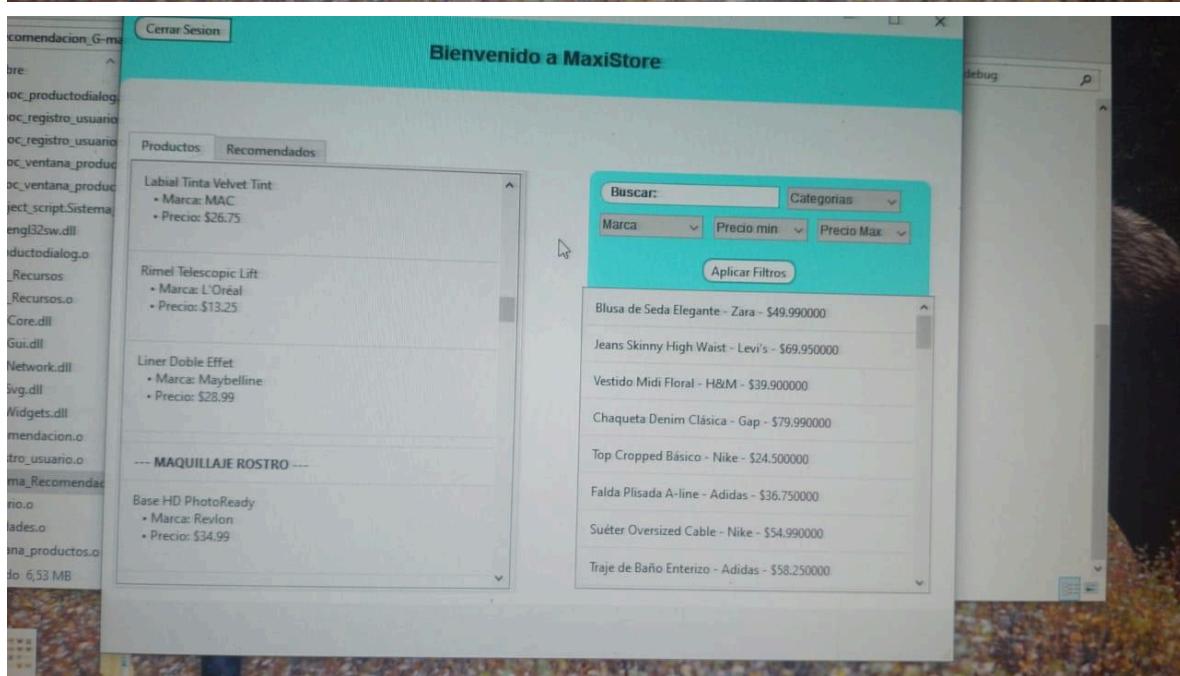
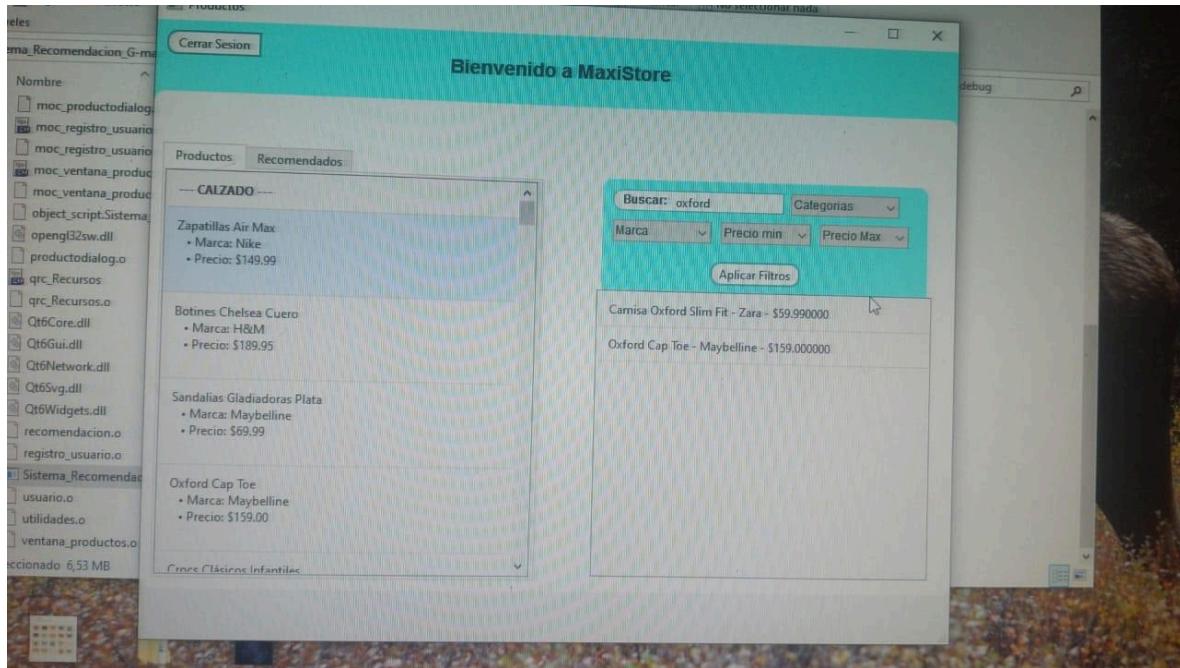


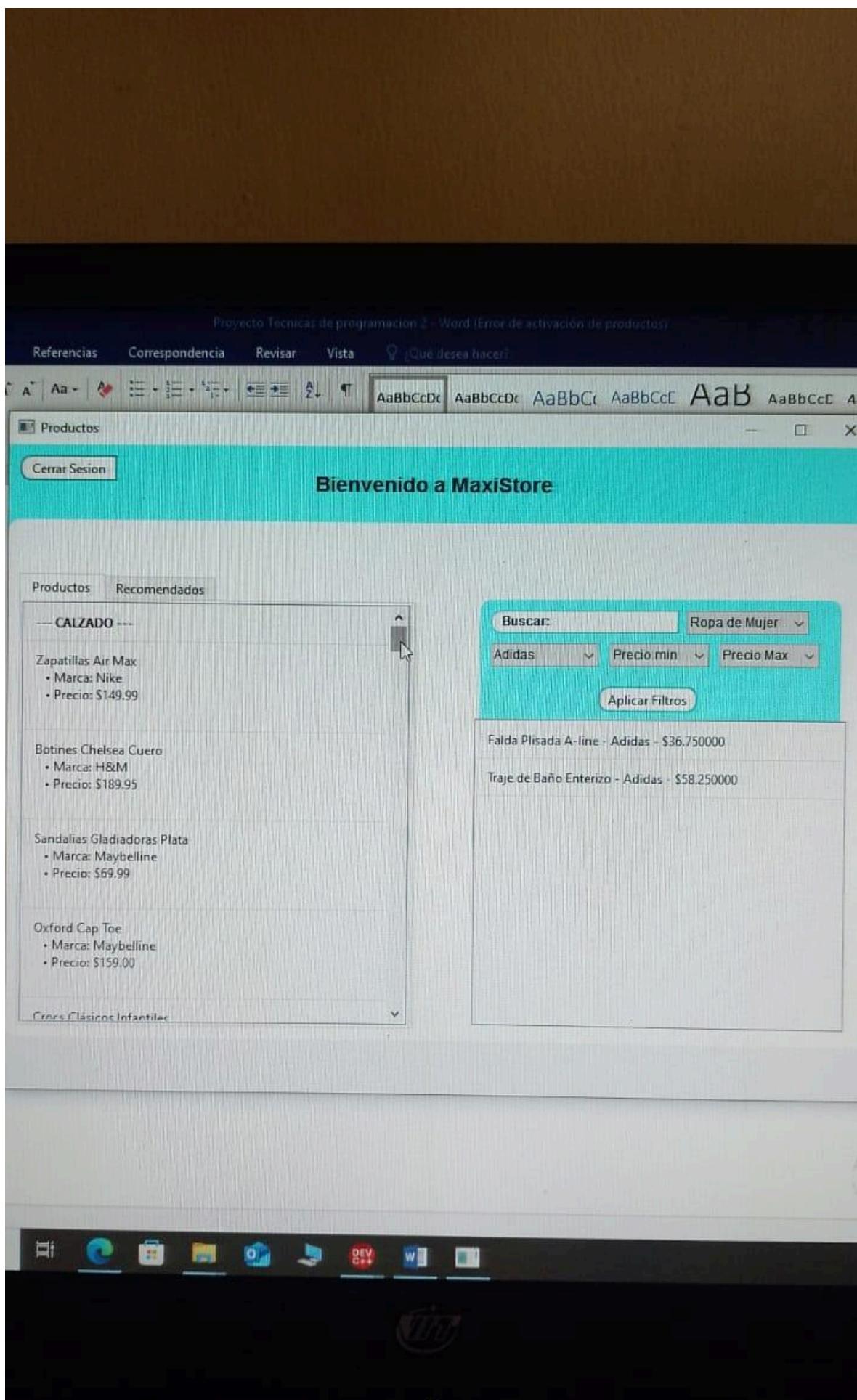


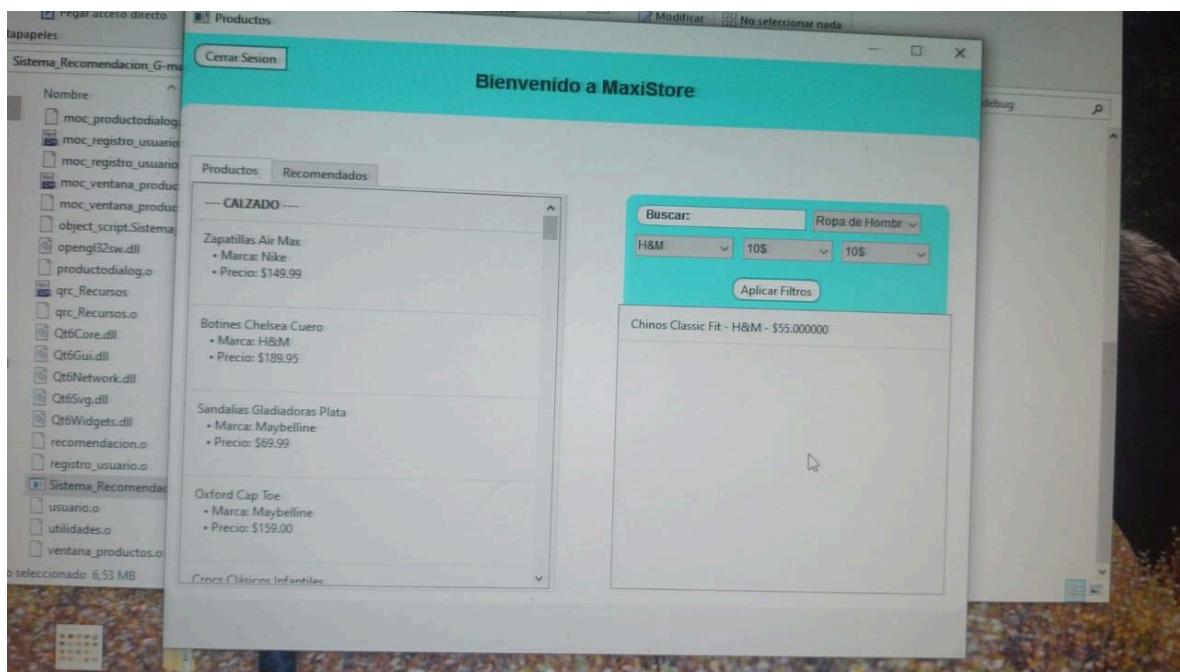


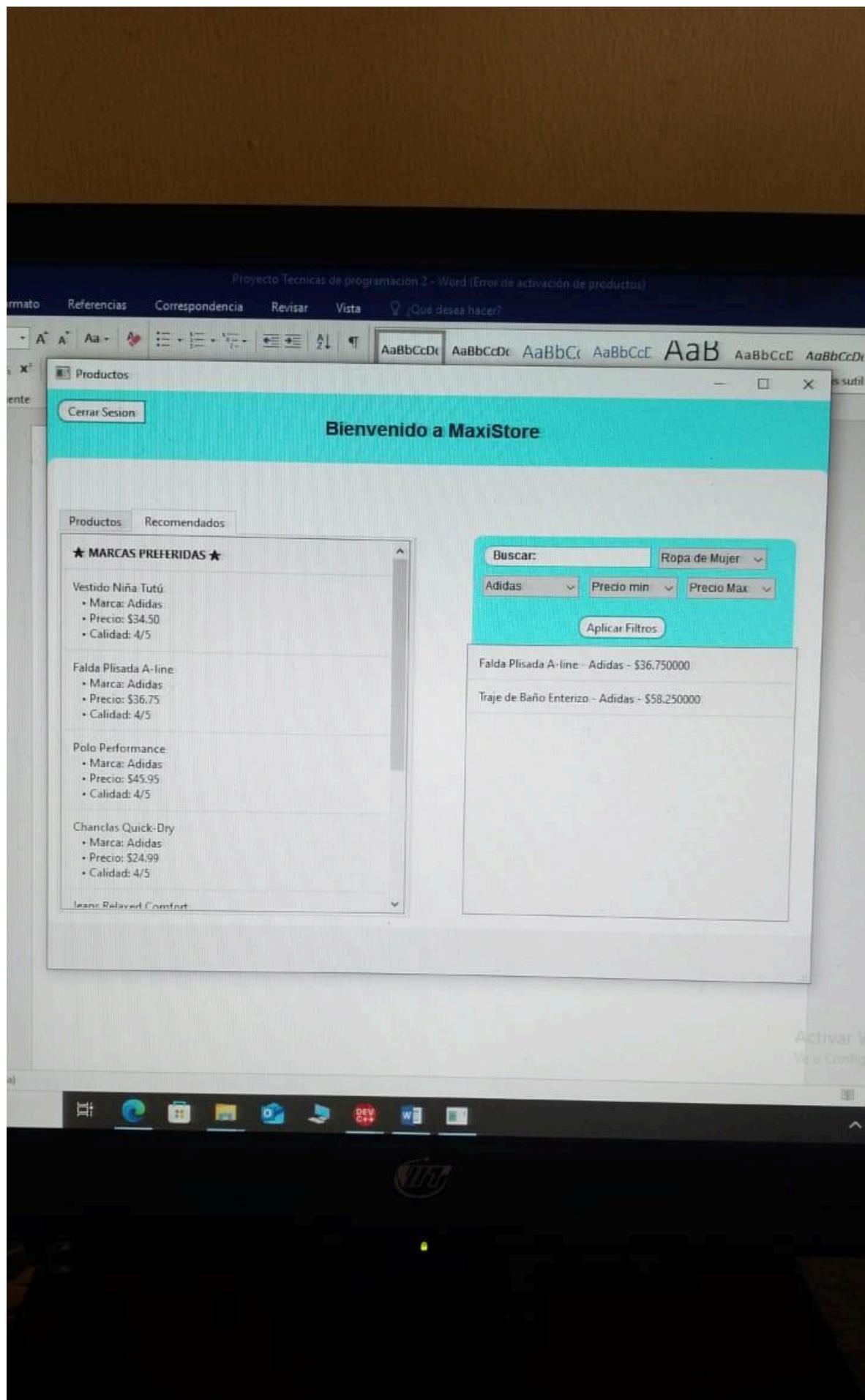


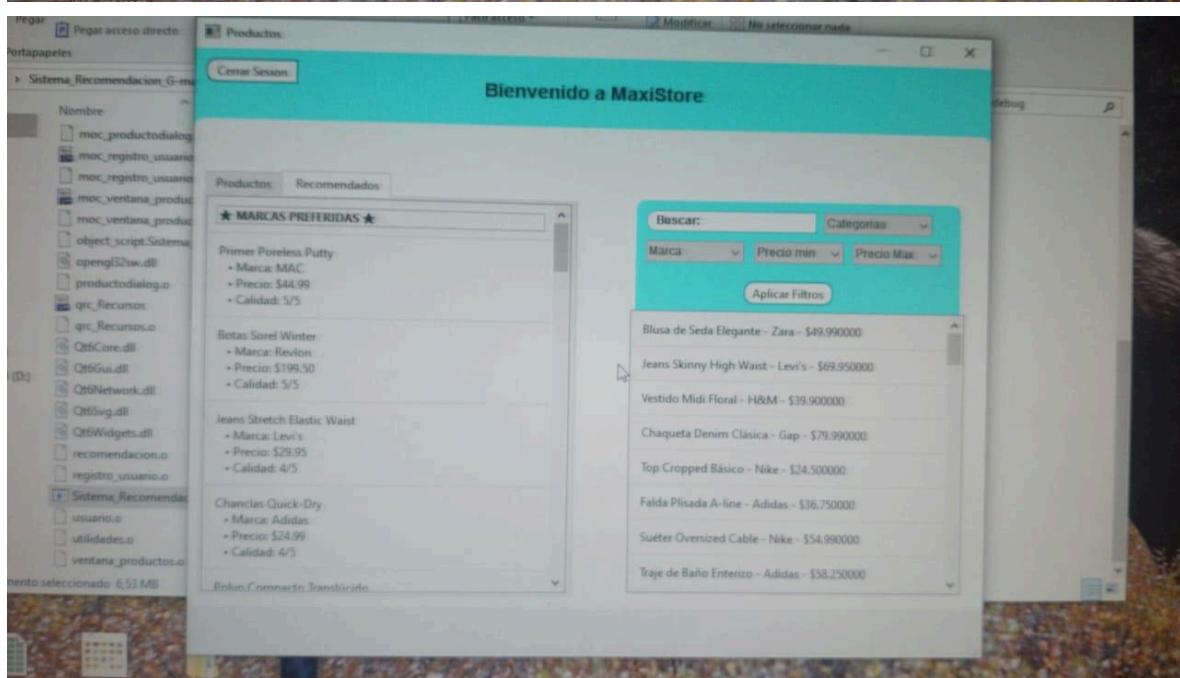
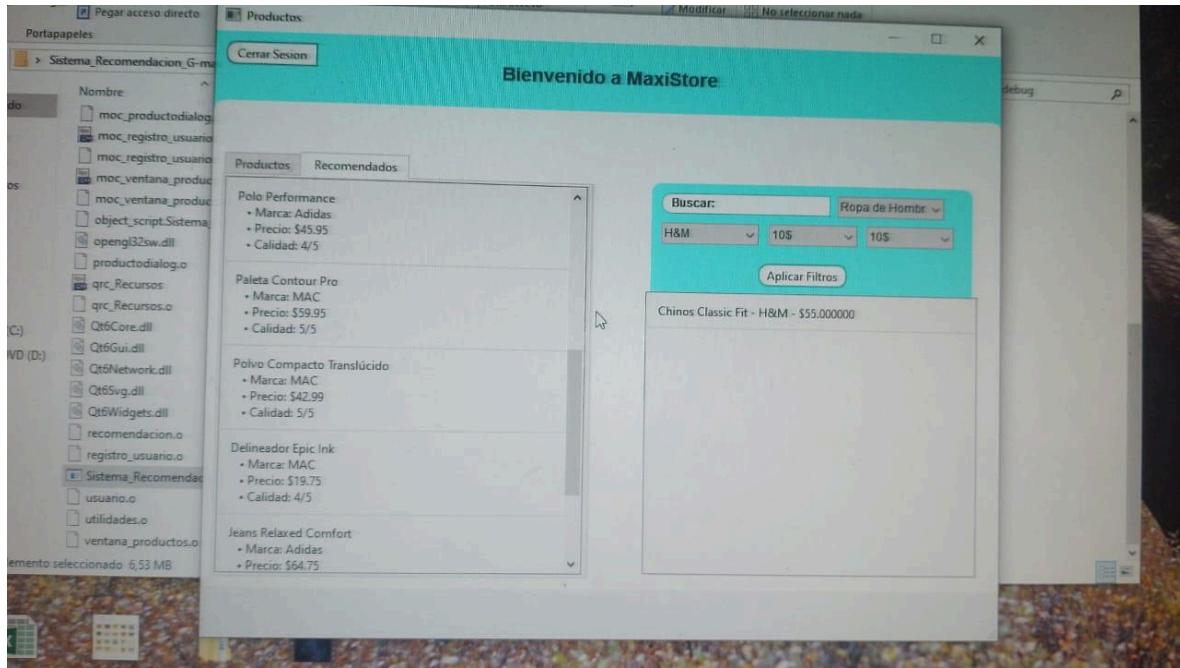


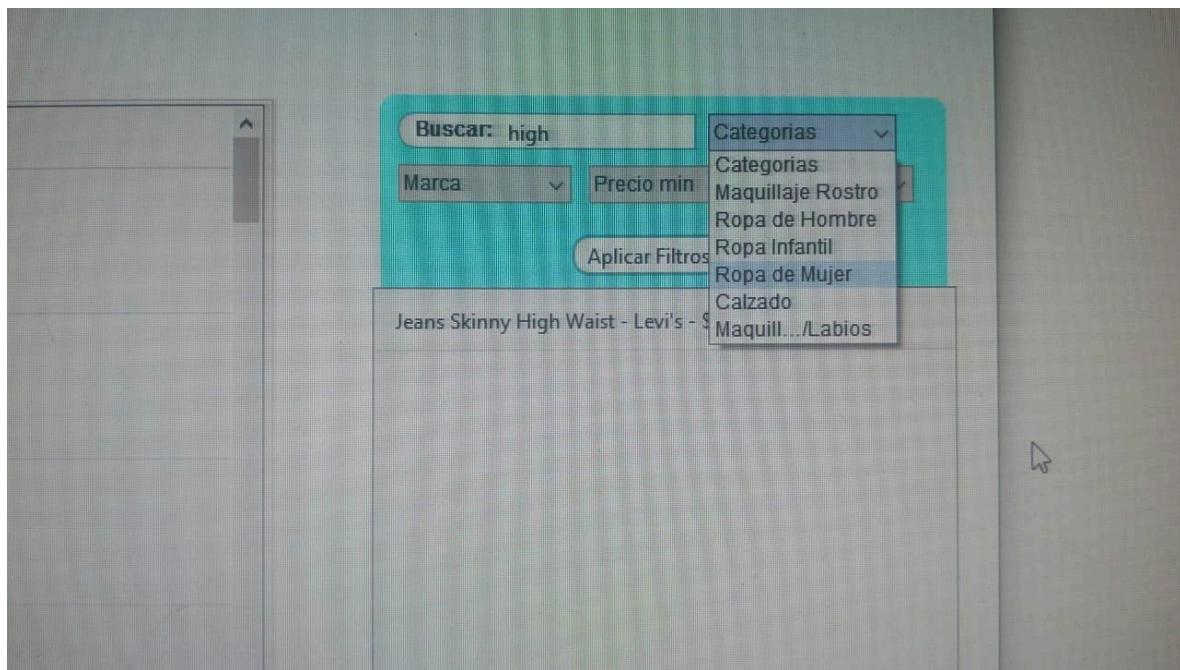
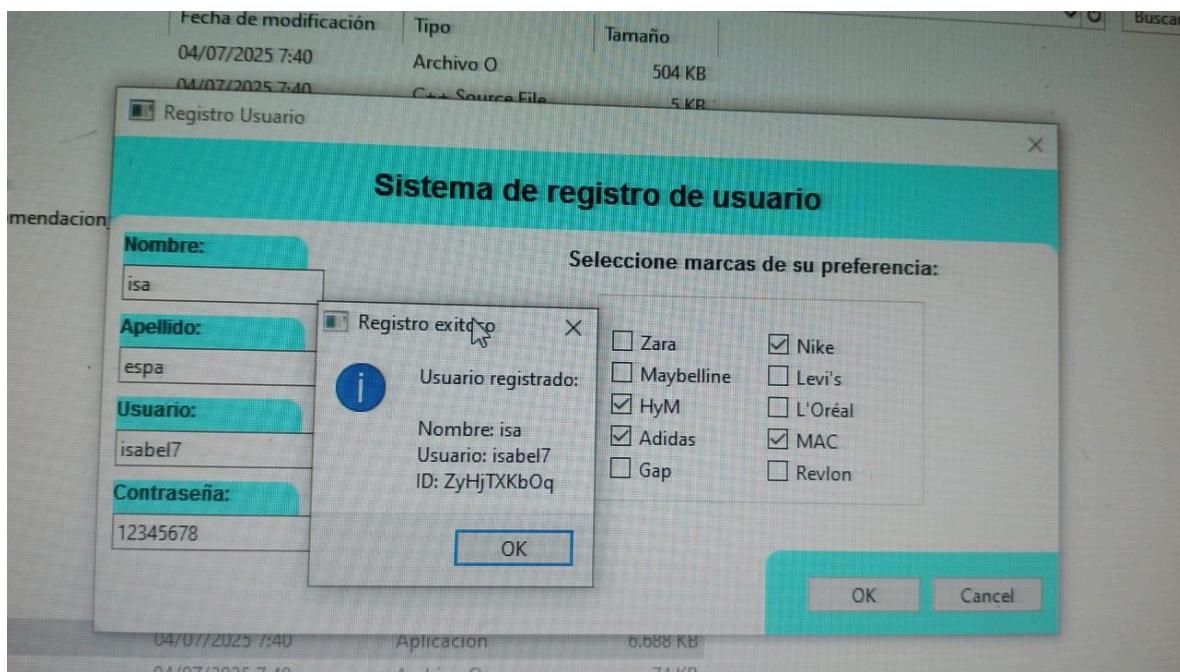


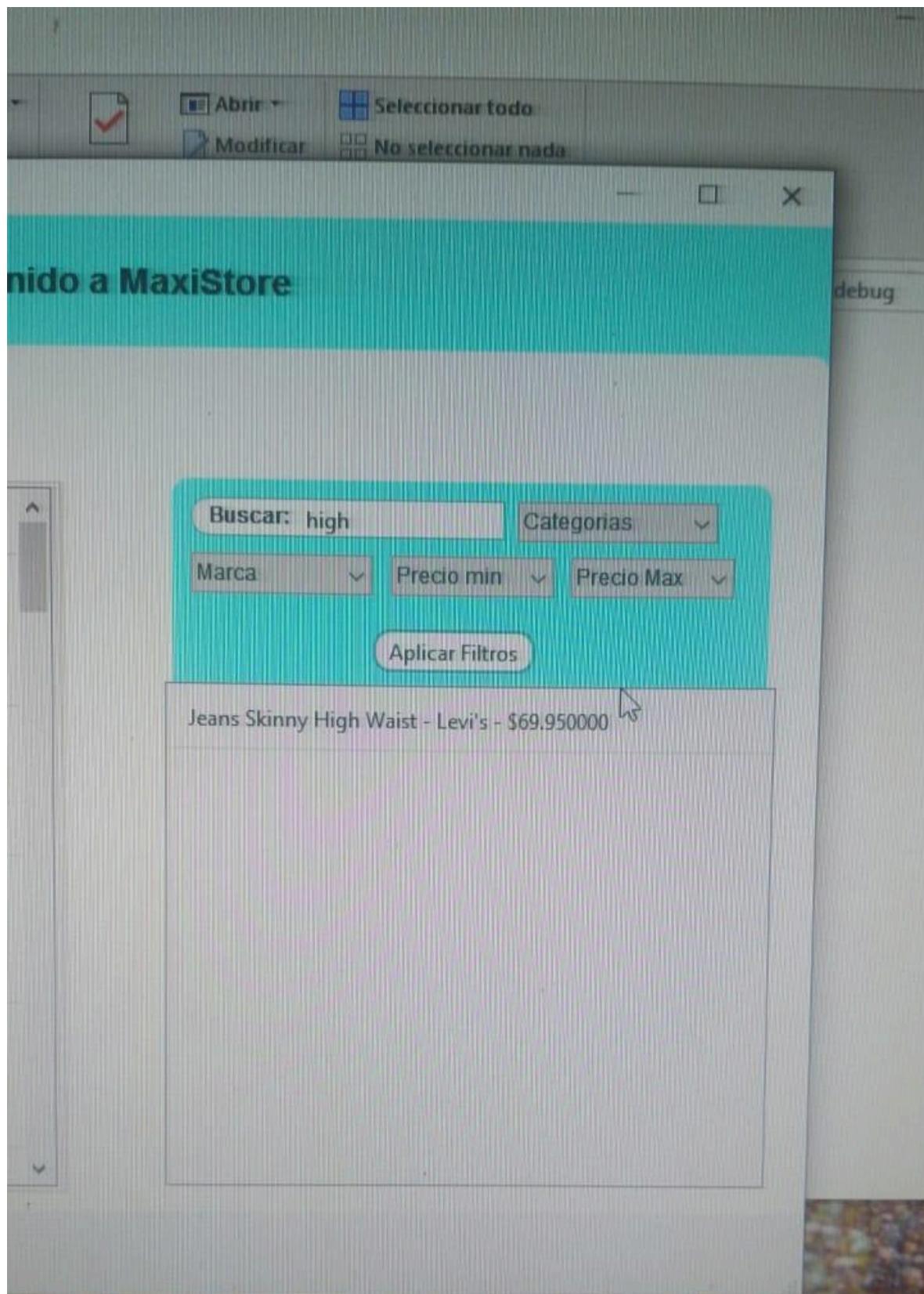


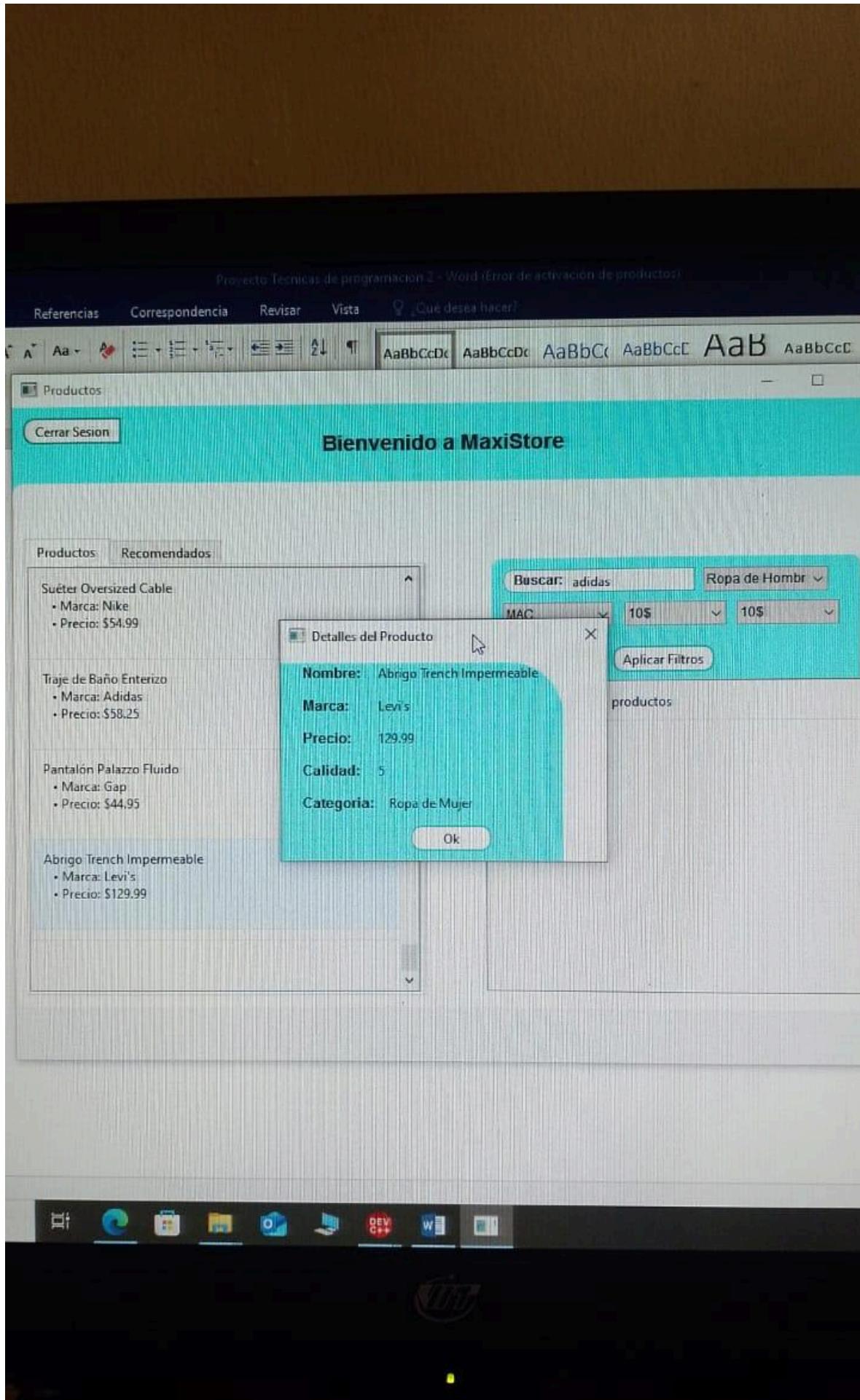




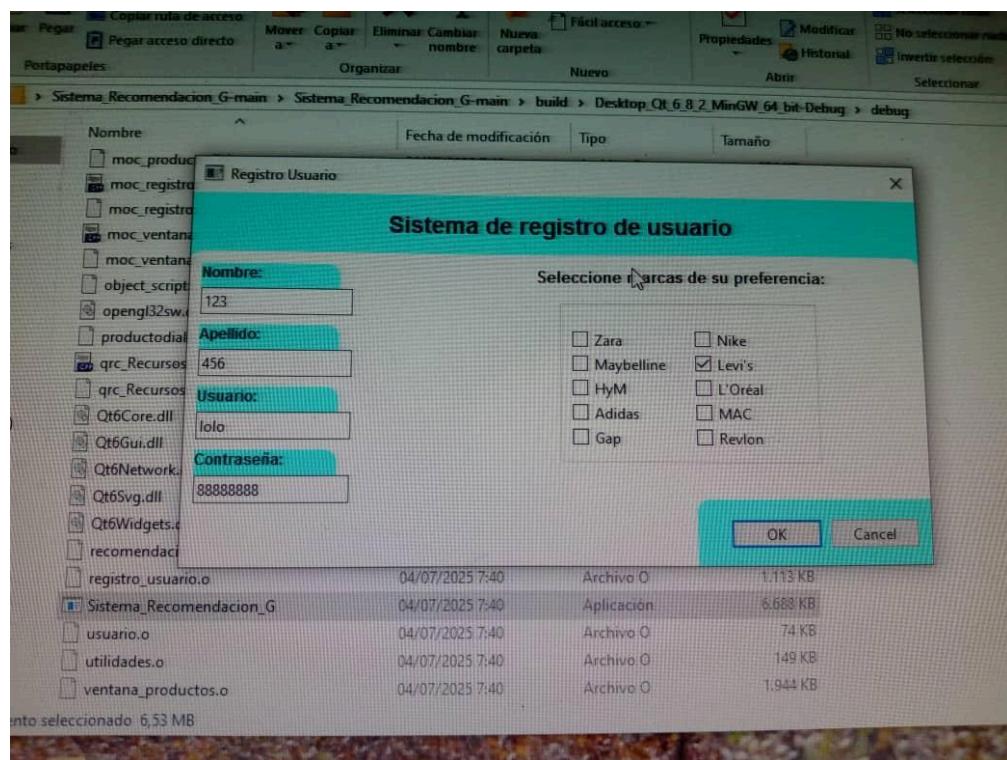


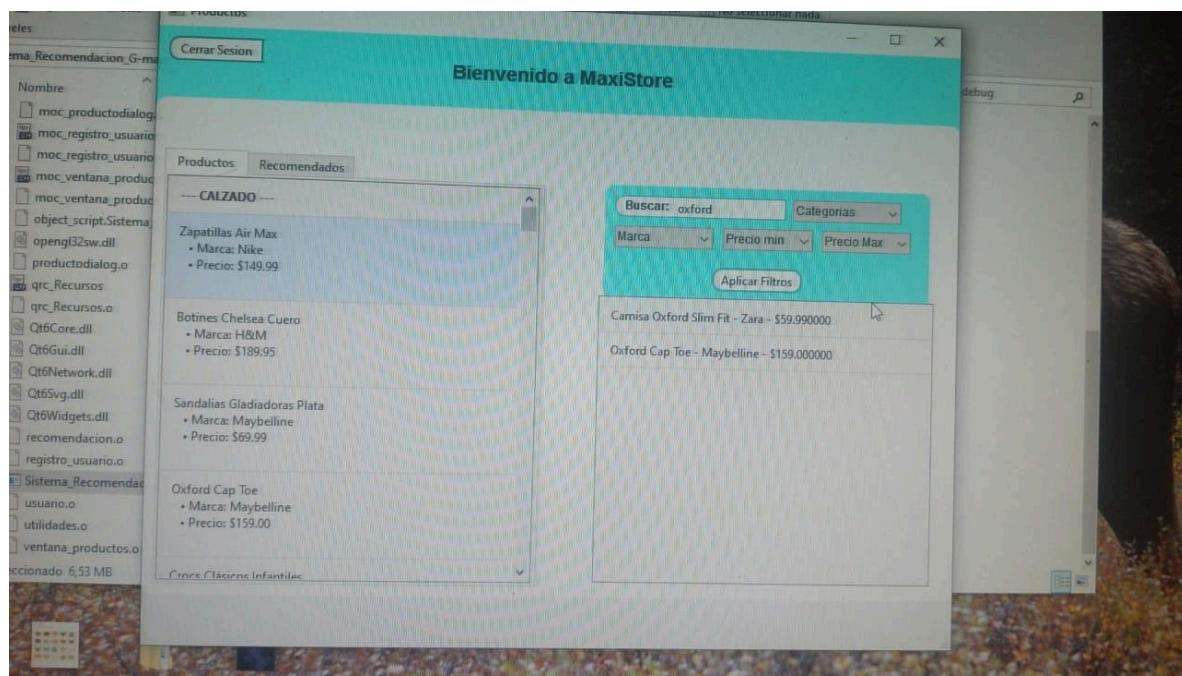
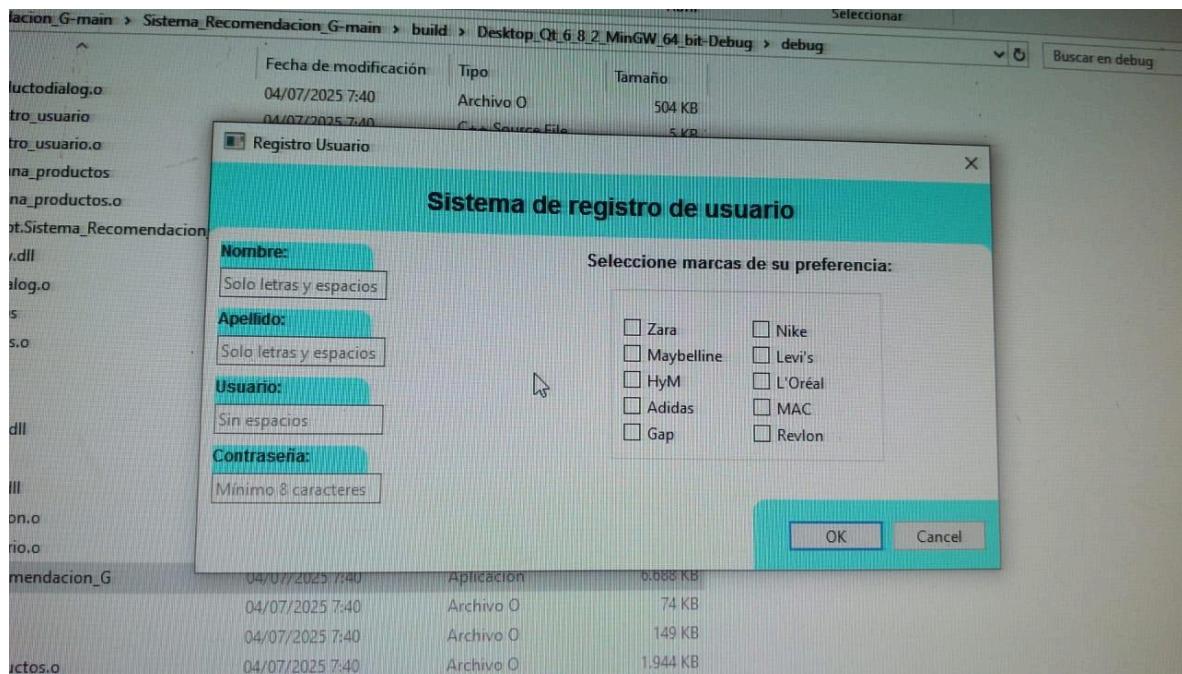


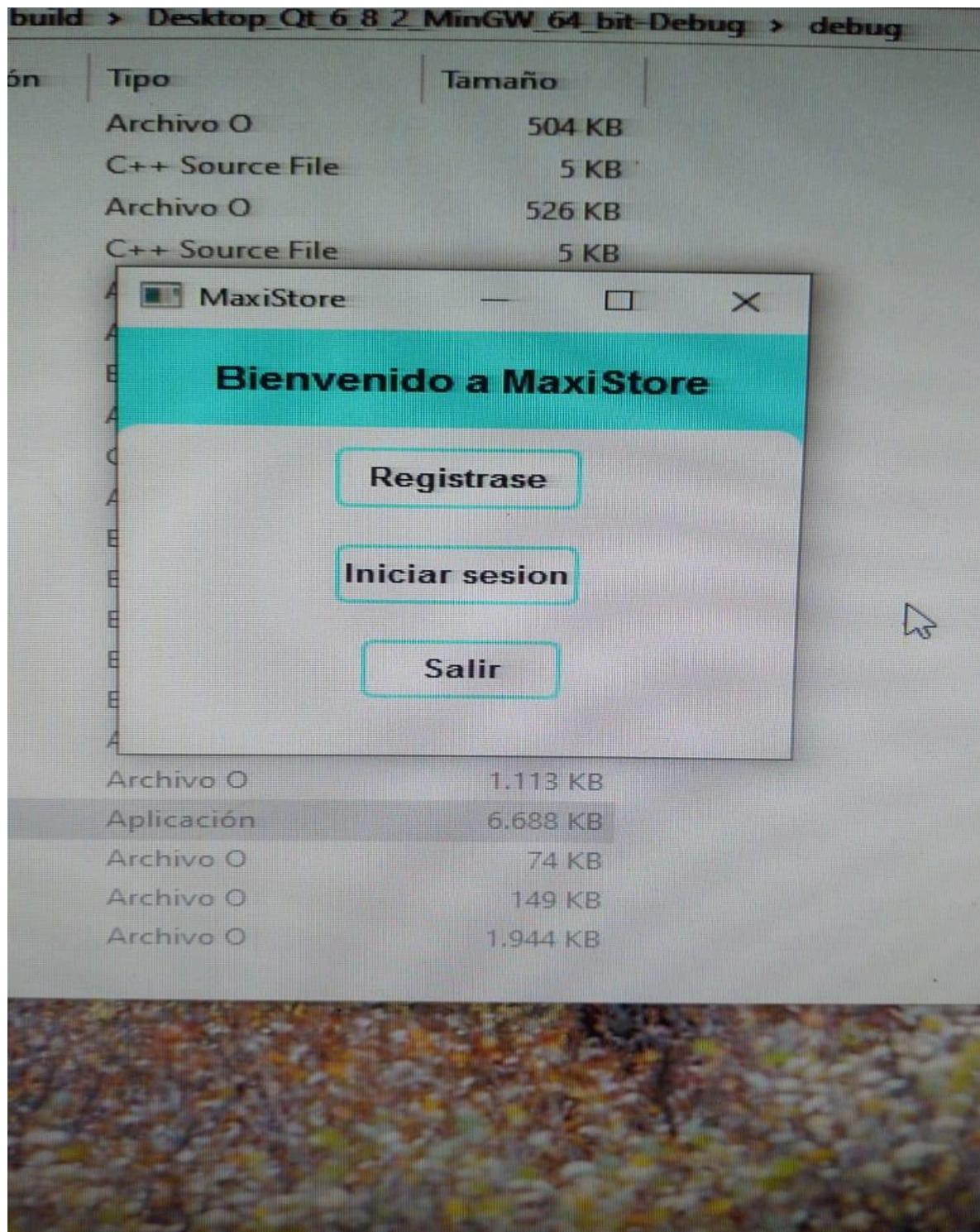


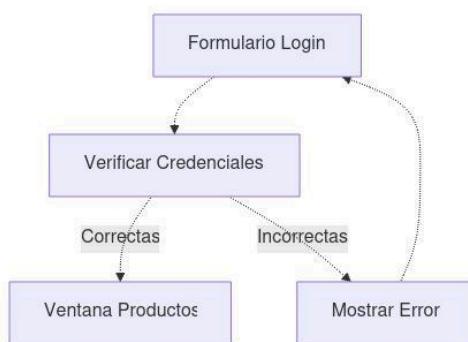
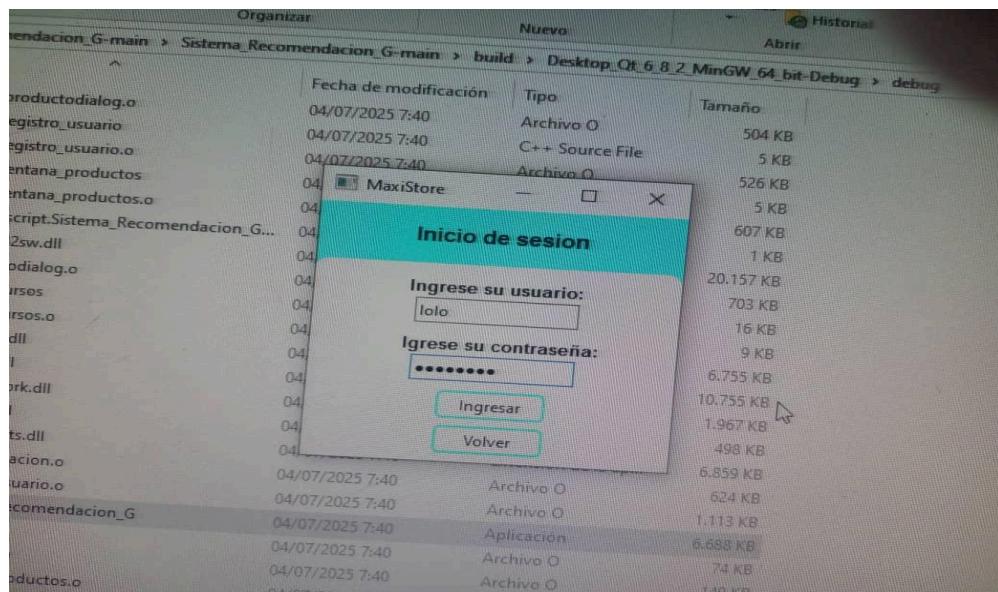


Ingreso de datos para crear usuario

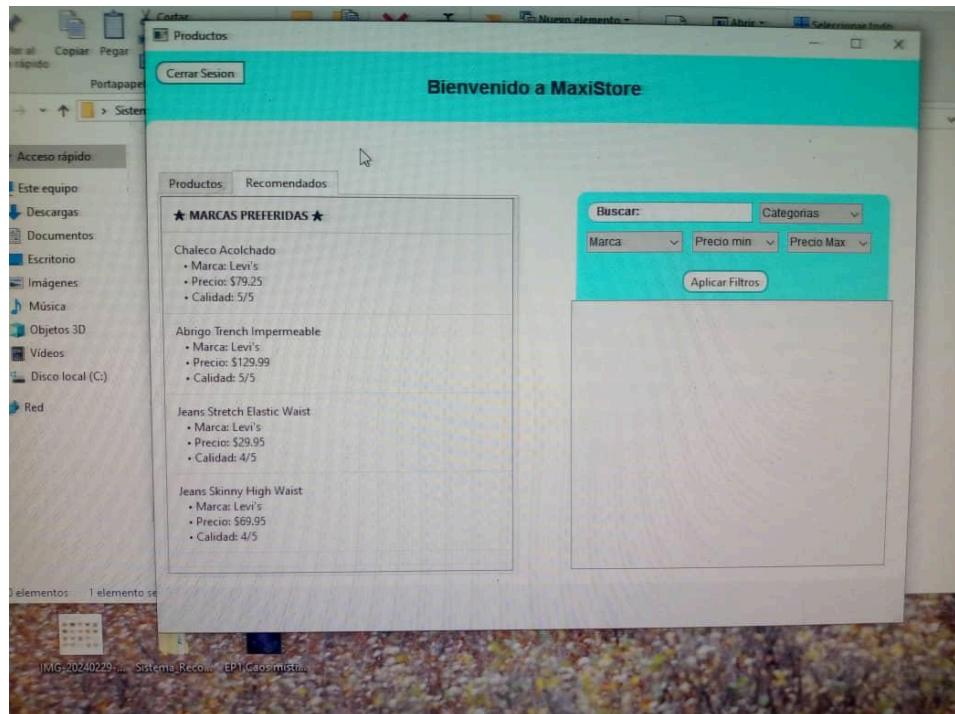


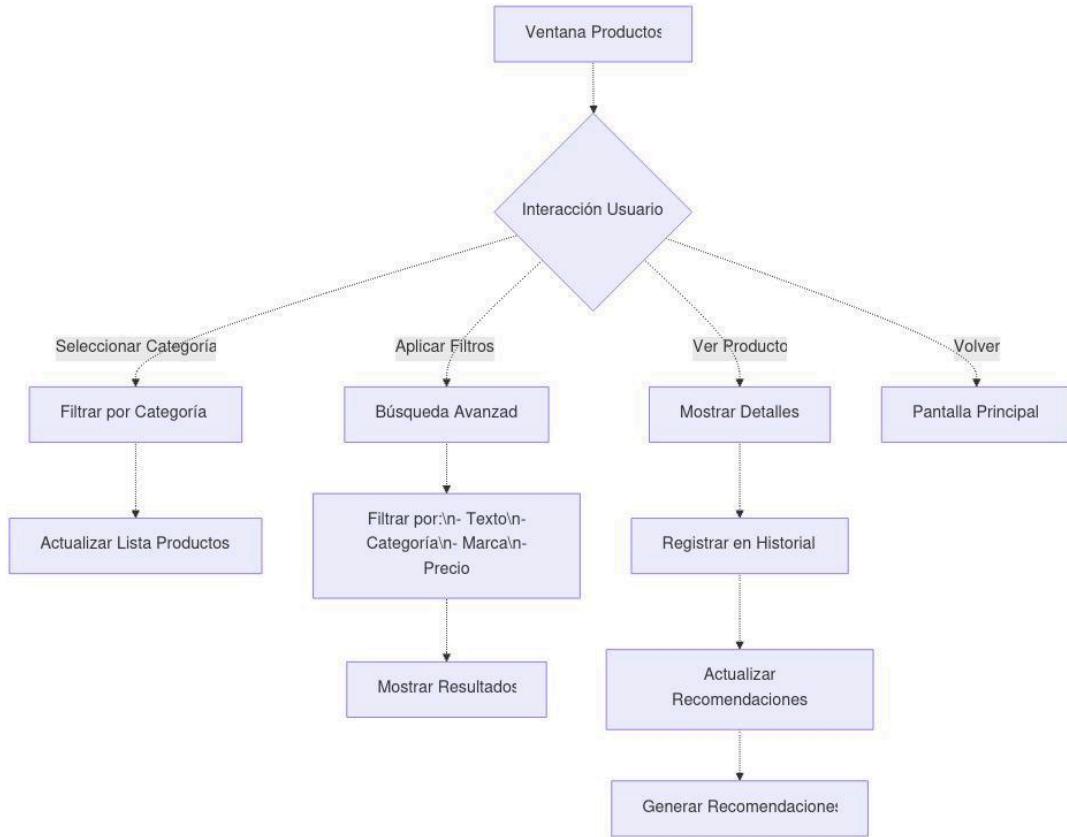




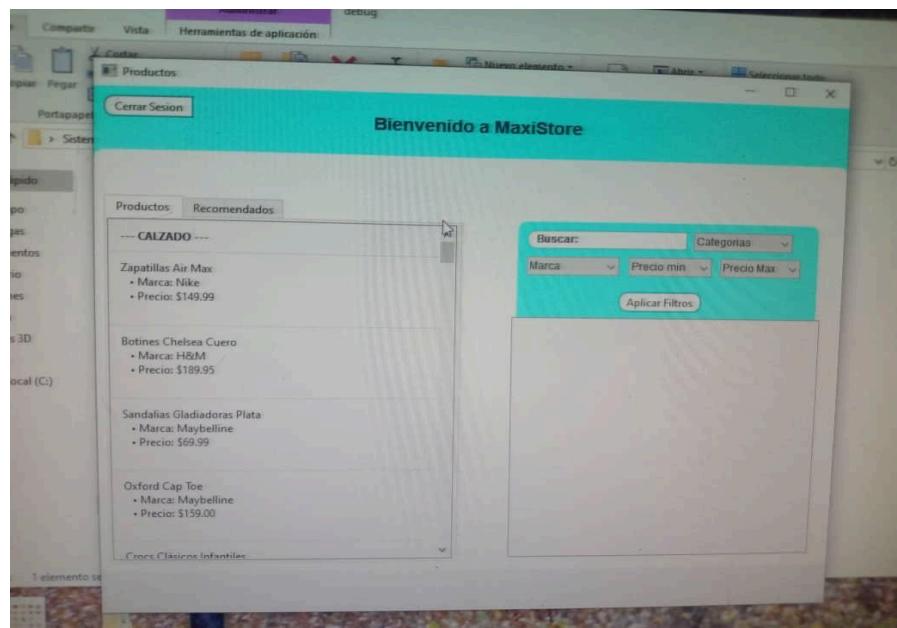


Catálogo de Producto

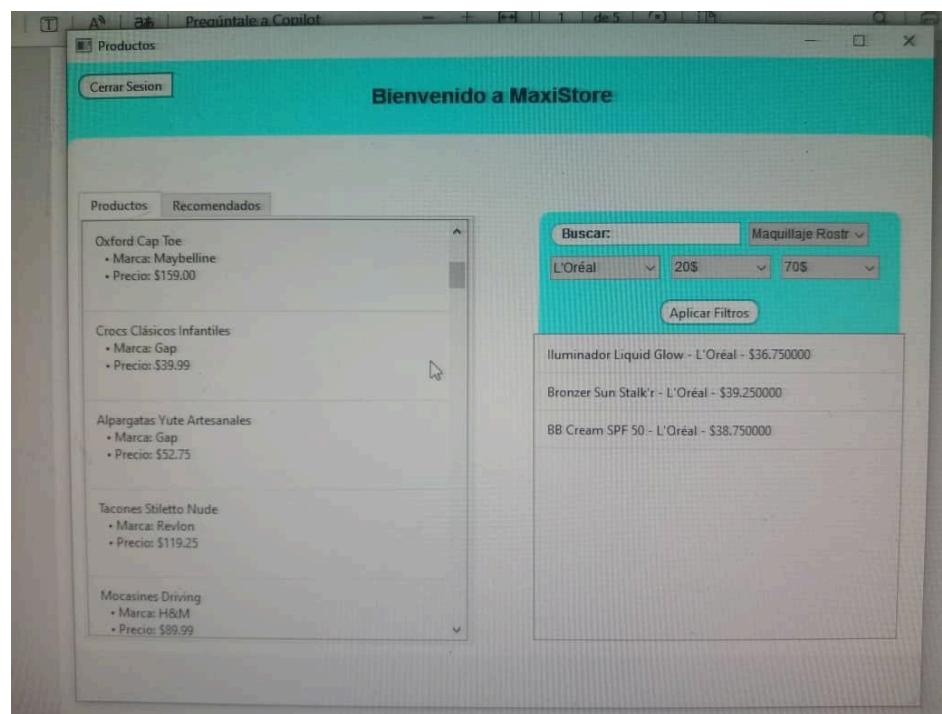




Opción de recomendados



Filtro de Productos



Información al seleccionar un producto

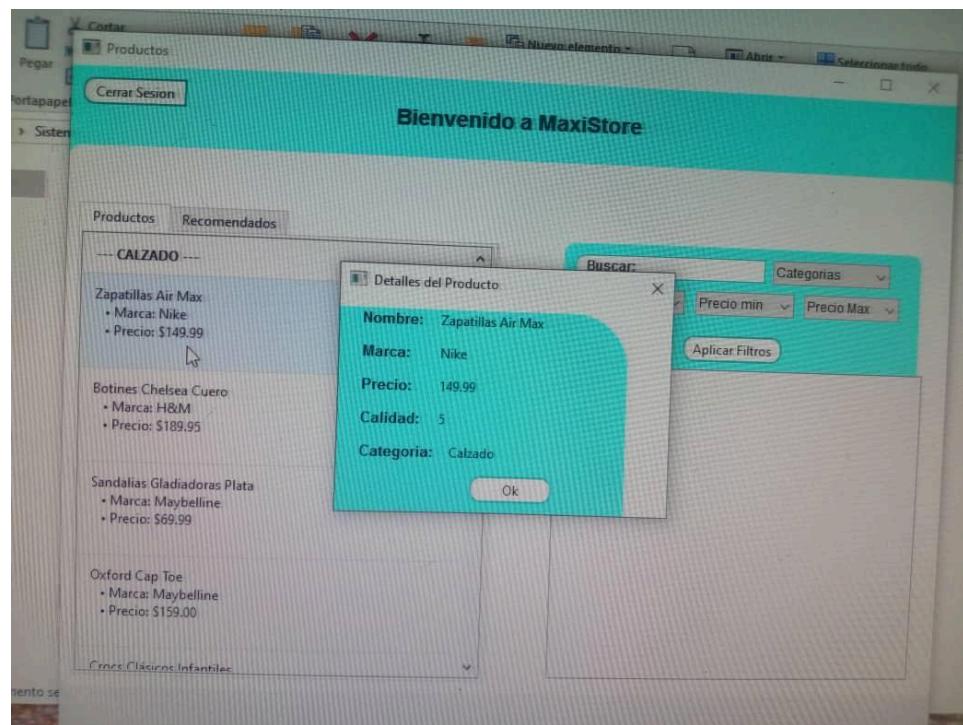
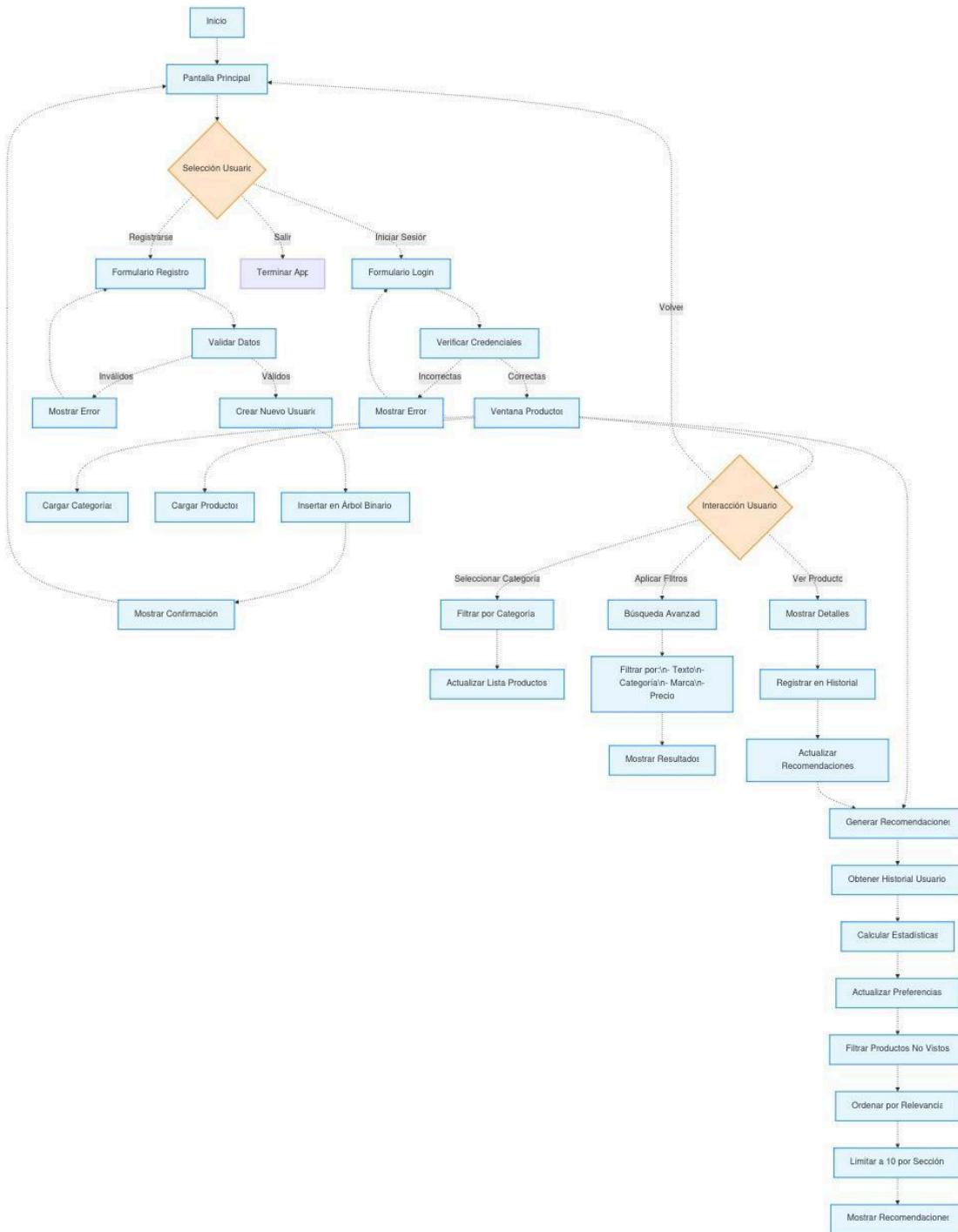


Diagrama de la lógica del programa



Código fuente:

catalogo.cpp

catalogo.h

estructuras.h

main.cpp

mainwindow.cpp

mainwindow.h

productodialog.cpp

productodialog.h

recomendacion.cpp

recomendacion.h

registro_usuario.cpp

recomendacion.h

registro_usuario.cpp

registro_usuario.h

usuario.cpp

usuario.h

utilidades.cpp

utilidades.h
ventana_productos.h
ventana_producto.cpp:

```
#include "ventana_productos.h"
#include "ui_ventana_productos.h"
#include "mainwindow.h"
#include "catalogo.h"
#include "recomendacion.h"
#include "productodialog.h"
#include <QListWidgetItem>
#include <QStringList>
#include <QDebug>
#include <QMessageBox>
#include <QRegularExpression>
#include <QIcon>
#include <QFont>
#include <QBrush>
#include <QColor>
#include <vector>
#include <algorithm>
```

//////////

/*para las cabeceras:

1. Modificador de acceso private:

Todo lo que está bajo la sección private: solo es accesible desde dentro de la misma clase. Esto significa que otras clases o funciones externas

no pueden acceder directamente a estos miembros o métodos.

2. Miembros privados de la clase

Estos son las variables y punteros que almacenan datos necesarios para el funcionamiento de la clase:

Ui::Ventana_Productos *ui;

Es un puntero a la interfaz gráfica generada automáticamente por Qt Designer (archivo .ui).

Este puntero permite acceder y manipular los elementos visuales de la ventana, como botones, etiquetas, listas, etc.

Usuario* usuarioActual;

Es un puntero a un objeto de tipo Usuario. Probablemente representa al usuario que está interactuando con la aplicación en ese momento de la preferencia

3. Métodos privados

Estos son las funciones que encapsulan la lógica interna de la clase. No son accesibles desde fuera de la clase, pero se usan para realizar

tareas específicas dentro de ella:

```
void cargarCategorias();
```

Este método probablemente carga las categorías de productos desde una base de datos, archivo o fuente externa.

```
void cargarProductosPorCategoria(const QString& categoria);
```

Este método recibe una categoría como parámetro (de tipo QString, una clase de cadena de texto en Qt).

Su función es cargar los productos que pertenecen a esa categoría específica, así como se puede actualizar en la tabla en la interfaz.

```
void cargarRecomendaciones();
```

Este método podría estar diseñado para cargar productos recomendados, basándose en las preferencias del usuario actual (usuarioActual)

durante el algoritmo de recomendacion.

```
void mostrarDetallesProducto(int id);
```

Este método recibe un identificador de producto (id) como parámetro.

Su propósito es mostrar los detalles del producto correspondiente, probablemente abriendo un cuadro de diálogo o actualizando una sección de la interfaz con detalles del producto.

/////////////////7su implementacion

diseñada para gestionar una ventana de productos en una aplicación qt, posee y maneja los datos privados, vease tal como: Un puntero a la interfaz gráfica (ui) y

Un puntero al usuario actual (usuarioActual). A su vez, hay métodos privados que son funciones para cargar categorías, productos por categoría, recomendaciones y mostrar detalles de un producto.

*//////////

extern ListaProducto catalogoGlobal;

/*1. extern: Declaración externa: La palabra clave extern se utiliza para declarar una variable o un objeto que está definido en otro archivo o

en otro lugar del programa. Esto le indica al compilador que no debe reservar memoria para esta variable, ya que su definición real está en otro

archivo fuente (generalmente en un archivo .cpp). donde se implemente para su llamado.

*/

//////////

Ventana_Productos::Ventana_Productos(Usuario* usuario,
QWidget *parent)

: QMainWindow(parent), ui(new Ui::Ventana_Productos),
usuarioActual(usuario)

{

ui->setupUi(this);

/*

constructor de una clase llamada Ventana_Productos, que se ejecuta automáticamente cuando se crea un objeto de la clase.

a. Ventana_Productos::Ventana_Productos

Esto indica que el constructor pertenece a la clase Ventana_Productos. El uso de :: (operador de resolución de ámbito)

b. Parámetros del Constructor

Usuario* usuario: Es un puntero a un objeto de tipo Usuario. Este parámetro probablemente se utiliza para pasar información del usuario actual a la ventana.

QWidget *parent: Es un puntero a un widget padre. En aplicaciones gráficas con Qt, esto se usa para establecer jerarquías entre ventanas

c. Lista de Inicialización

: QMainWindow(parent), ui(new Ui::Ventana_Productos),
usuarioActual(usuario)

La lista de inicialización se usa para inicializar atributos de la clase antes de que se ejecute el cuerpo del constructor.

/////////las inicializaciones surgen:

QMainWindow(parent): Llama al constructor de la clase base QMainWindow y le pasa el widget padre luego

ui(new Ui::Ventana_Productos): Crea una nueva instancia de la interfaz gráfica asociada a la clase Ventana_Productos para que

usuarioActual(usuario): es el que Inicializa el atributo usuarioActual con el puntero usuario recibido como parámetro.

el cuerpo del constructor:setupUi(this): Este método configura la interfaz gráfica de la ventana. Es generado automáticamente por el sistema

de diseño de Qt (Qt Designer) y conecta los elementos visuales (botones) para que el usuario interactue.

*/

/////////////////////////////aqui va mas orientado al diseño grafico de las ventanas el color y dimension del pixel.

/*

a partir de aqui, se personaliza la apariencia de un widget de lista (listWidget_3) en Qt. Definiendo:

Un color de fondo para el widget completo,un borde inferior y un relleno para cada elemento de la lista,un color de fondo especial para los elementos seleccionados.

```
*/  
  
ui->listWidget_3->setStyleSheet(  
    "QListWidget { background-color: #f8f8f8; }"  
    "QListWidget::item { border-bottom: 1px solid #e0e0e0;  
                        padding: 8px; }"  
    "QListWidget::item:selected { background-color:  
                                #e0f0ff; }"  
);
```

/* al desglosar el programa se tiene:

1. ui->listWidget_3

ui es un puntero al objeto que gestiona la interfaz gráfica de usuario (UI) en Qt. listWidget_3 es un widget específico del tipo QListWidget que probablemente fue creado en el archivo de diseño de la interfaz (un archivo .ui). Su uso para mostrar las lista de elementos

2. setStyleSheet()

nos documentmos que es un método de Qt que permite aplicar estilos personalizados a los widgets utilizando CSS-like syntax

(similar a las hojas de estilo en cascada de HTML). En este caso, se está definiendo el estilo visual de listWidget_3.

3. Cadenas de estilo (CSS-like syntax)

Dentro de setStyleSheet(), se pasa una cadena que define los estilos. Aquí se están configurando tres aspectos del QListWidget y sus elementos:

QListWidget { ... } Aplica estilos generales al widget completo, en este caso:

background-color: #f8f8f8; Establece un color de fondo claro para todo el widget.

QListWidget::item { ... } Aplica estilos a cada elemento individual de la lista, en este caso:

border-bottom: 1px solid #e0e0e0; Agrega una línea inferior gris claro entre los elementos.

padding: 8px; Añade un espacio interno (margen interno) de 8 píxeles alrededor del contenido de cada elemento.

`QListWidget::item:selected { ... }` Aplica estilos a los elementos seleccionados de la lista, en este caso:

`background-color: #e0f0ff;`: Cambia el color de fondo del elemento seleccionado a un azul claro.

4. Concatenación de cadenas: Las cadenas de estilo están concatenadas automáticamente porque están separadas por comillas consecutivas ("..." "..."").

* /

//////////Aqui se está configurando el estilo visual de un QListWidget en una interfaz gráfica creada con Qt en C++.

las estructuración se tiene:

1. Acceso al objeto listWidget:

ui->listWidget hace referencia al widget de tipo QListWidget que forma parte de la interfaz gráfica.

El prefijo ui indica que se trabajan con un archivo generado automáticamente por Qt Designer que posee elementos de la interfaz

2. Método setStyleSheet

setStyleSheet es un método de Qt que permite aplicar estilos personalizados a los widgets utilizando una sintaxis similar a CSS (Cascading Style Sheets).

Este método toma como argumento un string para definir los estilos.

3. Definición del estilo

El string que se pasa a setStyleSheet contiene varias reglas CSS específicas para el widget QListWidget y sus elementos.

Especificaciones al sistema:

a) Estilo general del QListWidget:

```
"QListWidget { background-color: #f8f8f8; }"
```

Aplica un color de fondo (background-color) al widget completo.

El color hexadecimal #f8f8f8 es un tono claro de gris.

b) Estilo de los elementos (item):

```
"QListWidget::item { border-bottom: 1px solid #e0e0e0;  
padding: 8px; }"
```

QListWidget::item se refiere a los elementos individuales dentro del QListWidget.

aplicandole la propiedad:

border-bottom: Agrega una línea inferior de 1 píxel con color gris claro (#e0e0e0) para separar los elementos.

padding: Agrega un espacio interno de 8 píxeles alrededor del contenido de cada elemento.

c) Estilo del elemento seleccionado:

```
"QListWidget::item:selected { background-color: #e0f0ff; }"
```

donde:

`QListWidget::item:selected` se aplica únicamente al elemento que está seleccionado en el evento

Propiedad que se le aplica al `background-color`: Cambia el color de fondo del elemento seleccionado a un tono azul claro (#e0f0ff).

//////////por lo tanto se tiene que al ejecutar el `QListWidget` tendrá:

Concatenación del estilo con todas las reglas de estilo que están concatenadas en un único string.

Cuando este código se ejecuta, Un fondo gris claro,elementos con líneas divisorias y un espacio interno,un color de fondo azul claro para el elemento seleccionado

*/

//////////

///Para personalizar el estilo de un widget gráfico (`QListWidget`) mediante una hoja de estilo (stylesheet). se tiene en la linea de comentario la explicacion:

`ui->listWidget_2->setStyleSheet(`

```
"QListWidget { background-color: #f8f8f8; }"  
"QListWidget::item { border-bottom: 1px solid #e0e0e0;  
    padding: 8px; }"  
"QListWidget::item:selected { background-color:  
    #e0f0ff; }"  
);  
/*  
definicion:
```

1. Contexto del estilo en el código:

El código modifica el estilo visual de un widget llamado `listWidget_2`, que es un objeto de tipo `QListWidget`.

Esto se hace mediante el método `setStyleSheet`, que permite aplicar estilos.

2. Desglosando el código se tiene:

a) `ui->listWidget_2`

`ui` es un puntero al objeto de interfaz de usuario generado automáticamente por `Qt Designer`.

`listWidget_2` es el nombre del widget específico que se está estilizando.(usando estilo)

b) setStyleSheet

Este método aplica una hoja de estilo en formato de cadena (string) al widget. La hoja de estilo sigue una sintaxis similar a CSS, pero adaptada a los widgets de Qt.

c) Hoja de estilo (stylesheet)

La cadena que se pasa a setStyleSheet contiene las reglas de estilo y las reglas son las siguientes:

```
QListWidget { background-color: #f8f8f8; }:
```

Aplica un color de fondo (background-color) al widget completo (QListWidget).

El color hexadecimal #f8f8f8 es un tono claro de gris.

```
QListWidget::item { border-bottom: 1px solid #e0e0e0;  
padding: 8px; }:
```

Aplica estilo a cada elemento individual (item) dentro del QListWidget.

border-bottom: 1px solid #e0e0e0;: Añade una línea inferior de 1 píxel con un color gris claro.

padding: 8px;: Añade un espacio interno de 8 píxeles alrededor del contenido del elemento.

QListWidget::item:selected { background-color: #e0f0ff; }:

Aplica estilo al elemento seleccionado (item:selected).

background-color: #e0f0ff;: Cambia el color de fondo del elemento seleccionado a un tono azul claro.

///////////77implementacion, el resultado visual esperado:

El QListWidget tendrá un fondo gris claro.

Cada elemento tendrá una línea inferior gris y un espacio interno de 8 píxeles.

Cuando un elemento sea seleccionado, su fondo cambiará a azul claro.

*/

//////////

//Aquí se establece una conexión entre una señal y un slot utilizando una funcion lambda

```
connect(ui->listWidget_3, &QListWidget::itemClicked,
       [this](QListWidgetItem* item) {
    if (item->flags() & Qt::ItemIsSelectable) { // Solo si es
        seleccionable (no encabezado)
        int id = item->data(Qt::UserRole).toInt();
        mostrarDetallesProducto(id);
    }
});
```

descripción del proceso y como se estructura:

1) connect:

Es una función de Qt que conecta una señal (signal) con un slot (función que responde a la señal).

En este caso, se conecta la señal itemClicked del widget listWidget_3 con una lambda function

2) ui->listWidget_3:

Hace referencia a un widget de tipo QListWidget que está definido en la interfaz gráfica (UI).

Este widget es una lista que puede contener elementos (QListWidgetItem).

3)&QListWidget::itemClicked:

Es la señal que emite el widget `listWidget_3` cuando un elemento de la lista es clicado.

Esta señal pasa como argumento un puntero al elemento clicado (`QListWidgetItem*`).

4)Lambda function:

`[this](QListWidgetItem* item) { ... }` es una función anónima (lambda) que se ejecuta cuando se emite la señal `itemClicked`.

`[this]`: Captura el puntero `this` para que la lambda pueda acceder a los miembros de la clase actual.

`(QListWidgetItem* item)`: Define el parámetro que recibe la lambda, en este caso, el elemento clicado.

5)`item->flags()` & `Qt::ItemIsSelectable`:

`item->flags()` devuelve los "flags" (propiedades) del elemento clicado.

`Qt::ItemIsSelectable` es un flag que indica si el elemento puede ser seleccionado.

La condición verifica si el elemento es seleccionable antes de continuar.

6) item->data(Qt::UserRole).toInt():
item->data(Qt::UserRole) obtiene un dato asociado al elemento, almacenado bajo el rol Qt::UserRole.
.toInt() convierte ese dato a un entero. Este entero se interpreta como el identificador (id) del producto.

7) mostrarDetallesProducto(id):
Es una función de la clase actual que se llama con el identificador del producto (id). Su propósito es mostrar los detalles del producto

//////////por lo tanto, en la implementacion:
en el código, se conecta la acción de hacer clic en un elemento de la lista (itemClicked) con una función lambda que verifica si el elemento es seleccionable, si obtiene un identificador asociado al elemento y llama a una función (mostrarDetallesProducto) para mostrar los detalles del producto que corresponda.

```
*/  
///////////Aquí se está configurando una  
conexión entre una señal y un bloque de código (lambda)  
con usar qt
```

```
connect(ui->listWidget, &QListWidget::itemClicked,  
[this](QListWidgetItem* item) {  
    if (item->flags() & Qt::ItemIsSelectable) {  
        int id = item->data(Qt::UserRole).toInt();  
        mostrarDetallesProducto(id);  
    }  
/*
```

las funcionalidades :

1. connect:

La función connect es utilizada en Qt para conectar señales y slots. En este caso, se conecta una señal emitida por un widget

(en este caso, un QListWidget) con un slot (un bloque de código que se ejecutará cuando la señal sea emitida).
interaccion con el usuario e interfaz

2. Parámetros de connect:

ui->listWidget: Es el objeto que emite la señal. Aquí se refiere a un QListWidget

&QListWidget::itemClicked: Es la señal que se emite cuando un elemento de la lista (QListWidgetItem) es clicado.

pasa esta señal como argumento un puntero al elemento clicado.

[this](QListWidgetItem* item) { ... }: Es un lambda (una función anónima) que actúa como el slot. Este lambda se ejecutará cuando la señal itemClicked se emita. lo siguiente representa:

[this]: Captura el puntero this (la instancia actual de la clase) para que se pueda acceder dentro del lambda.

QListWidgetItem* item: Es el parámetro que recibe el lambda. Representa el elemento de la lista que fue clicado por el usuario.

//////////

la logica con el lambda cuando se le da click o se quiere llegar a elementos seleccionables:

item->flags() & Qt::ItemIsSelectable: Se verifica si el elemento clicado tiene la bandera Qt::ItemIsSelectable.

item->data(Qt::UserRole).toInt(): Se obtiene un dato asociado al elemento usando la clave Qt::UserRole. Este dato es convertido a un entero (toInt()).

3.mostrarDetallesProducto(id):

Se llama a un método de la clase (mostrarDetallesProducto) pasando el identificador (id) como argumento. Este método probablemente muestra información detallada del producto que se asocia a un elemento clicableado al evento.

/////////////////7 por lo tanto en esta sección de código la funcionalidad sería:

cuando se conecta la señal itemClicked de un QListWidget con un lambda sucede que se verifica si el elemento clicado es seleccionable, se

obtiene un identificador asociado al elemento y llama a un método para mostrar detalles (características calida..) del producto correspondiente.

*/

});

/////////// utiliza conexiones de señales y slots,
un mecanismo central en Qt

```
connect(ui->listWidget_2, &QListWidget::itemClicked,  
[this](QListWidgetItem* item) {//Primera conexión: connect  
    con una lambda  
  
    int id = item->data(Qt::UserRole).toInt();  
  
    mostrarDetallesProducto(id);  
  
});
```

```
connect(ui->pushButton, &QPushButton::clicked, this,  
&Ventana_Productos::on_pushButton_clicked);
```

/* conexión con lambda:

connect: Es una función de Qt que conecta una señal (evento) con un slot (función que responde al evento).

ui->listWidget_2: Es un puntero al widget de tipo QListWidget que está en la interfaz gráfica. Aquí se conecta su señal itemClicked.

&QListWidget::itemClicked: Es la señal emitida cuando un elemento de la lista (QListWidgetItem) es clicado.

Lambda [this](QListWidgetItem* item): Es una función anónima (lambda) que se ejecuta cuando la señal itemClicked es emitida.

[this]: Captura el puntero this para acceder a los miembros de la clase actual.

(QListWidgetItem* item): Recibe como argumento el elemento clicado(seleccionado con click)

cuerpo de lambda: int id =
item->data(Qt::UserRole).toInt();mostrarDetallesProducto(id
);

item->data(Qt::UserRole): Obtiene un dato asociado al elemento clicado, almacenado bajo el rol Qt::UserRole.

.toInt(): Convierte ese dato a un entero.// hace la conversion pero empleando con qt

mostrarDetallesProducto(id): Llama a una función miembro de la clase actual, pasando el ID del producto para mostrar sus detalles.

2.Segunda conexión: connect con un slot miembro:
`connect(ui->pushButton, &QPushButton::clicked, this,
&Ventana_Productos::on_pushButton_clicked);`

solo sucede si es click:

ui->pushButton: Es un puntero al botón de la interfaz gráfica.

&QPushButton::clicked: Es la señal emitida cuando el botón es clicado.

this: Es el puntero a la instancia actual de la clase (probablemente Ventana_Productos).

&Ventana_Productos::on_pushButton_clicked:

Es un puntero a un método miembro de la clase Ventana_Productos. con el click del usuario.

/////////////////////////////por lo tanto:

con la primera conexión, usa una lambda para definir el comportamiento directamente en línea.

con la segunda conexión se conecta directamente una señal con un slot miembro predefinido.

aquí se conecta eventos de la interfaz gráfica con funciones que manejan esos eventos tales como:

casos de hacer clic en un elemento de la lista (listWidget_2), se ejecuta una lambda que obtiene un ID y llama a mostrarDetallesProducto.

caso de hacer clic en un botón (pushButton), se ejecuta el método on_pushButton_clicked de la clase Ventana_Productos.

*/

///////////////////////////////7
//realiza ciertas tareas específicas relacionadas con la gestión de datos, como llenar filtros, cargar productos por categoría y cargar recomendaciones.

llenarFiltros();

cargarProductosPorCategoria("Todas las categorías");
cargarRecomendaciones();
}

/*////////////////////////funcionalidades del codigo:

llenarFiltros(); :

Es una llamada a una función llamada llenarFiltros. y

Su propósito parece ser inicializar o llenar los filtros que se usarán en el programa, posiblemente para filtrar productos, categorías u otros datos.

cargarProductosPorCategoria("Todas las categorías");

Es una llamada a una función llamada
cargarProductosPorCategoria.

Esta función recibe un parámetro de tipo string (en este caso, "Todas las categorías"), lo que indica que probablemente carga productos relacionados

con una categoría específica. así como la interaccion de lista para obtener y mostrar los productos.

cargarRecomendaciones();

Es una llamada a una función llamada
cargarRecomendaciones.

No recibe parámetros, lo que sugiere que genera recomendaciones basadas en datos preexistentes, en base a preferencias del usuario, y productos recomendados

*/

/* por lo tanto , el objetivo tiende a ser como :

Aplicaciones de comercio electrónico: Para mostrar productos y recomendaciones.(algoritmo para negocios)

Sistemas de gestión: Para filtrar y cargar datos según categorías.(tipos de marcas)

*/

///////////////////////////////Función: llenarFiltros

//

/*

Esta función pertenece a la Ventana_Productos y tiene como propósito llenar un QComboBox (un desplegable en la interfaz gráfica)

con categorías únicas de productos obtenidas de una lista enlazada de productos (NodoProducto).

*/

void Ventana_Productos::llenarFiltros()

{

ui->comboBox->clear();

ui->comboBox->addItem("Categorias");

```
NodoProducto* actualCat = catalogoGlobal; //Se recorre la lista enlazada de productos usando el puntero actualCat.
```

```
QSet<QString> categoriasSet;//Uso de QSet: Es una elección eficiente para manejar categorías únicas,
```

```
while (actualCat != nullptr) { //Recorrido de la lista enlazada:while
```

```
//Interfaz gráfica (ui): El prefijo ui-> indica que el programa utiliza un archivo de interfaz gráfica
```

```
//generado por Qt (como un archivo .ui), y comboBox es un elemento de dicha interfaz
```

```
//
```

```
categoriasSet.insert(QString::fromStdString(actualCat->dato.categoría));
```

```
actualCat = actualCat->siguiente;//Luego, actualCat avanza al siguiente nodo de la lista enlazada
```

```
} //Recorrido de la lista enlazada:while
```

```
for (const QString& categoria : categoriasSet) {
```

```
    ui->comboBox->addItem(categoría);
```

```
/*
```

Se recorre el QSet (categoriasSet) que contiene las categorías únicas.

Cada categoría se agrega como un nuevo elemento al QComboBox mediante addItem.

```
 */  
}  
/*
```

estructuras paso apaso:

1)Limpieza del QComboBox: ui->comboBox->clear();

Antes de agregar nuevos elementos al QComboBox, se eliminan los elementos existentes

2)Agregar un elemento inicial:

```
ui->comboBox->addItem("Categorias");
```

Se agrega un elemento inicial al QComboBox con el texto "Categorias". es el titulo texto central

3)Inicialización de variables: NodoProducto* actualCat = catalogoGlobal; QSet<QString> categoriasSet;

actualCat es un puntero que se inicializa con el inicio de la lista enlazada de productos (catalogoGlobal).

categoriasSet es un contenedor de tipo QSet<QString>, que se utiliza para almacenar categorías únicas.

En cada iteración:

Se convierte la categoría del producto (que está en formato std::string) a QString usando QString::fromStdString.

La categoría convertida se inserta en el QSet (categoriasSet).

//////////por lo tanto la funcionalidad seria:

Limpia el QComboBox, Agrega un elemento inicial ("Categorias"), Recorre la lista enlazada de productos (NodoProducto), Extrae las categorías y las almacena en un QSet para evitar duplicados.

y Agrega las categorías únicas al QComboBox.

*/

//////////7

ui->comboBox_2->clear();

ui->comboBox_2->addItem("Marca");

NodoProducto* actualMarca = catalogoGlobal;

QSet<QString> marcasSet;

while (actualMarca != nullptr) {

```
marcasSet.insert(QString::fromStdString(actualMarca->dato.  
marca));  
  
actualMarca = actualMarca->siguiente;  
}//Recorre la lista enlazada catalogoGlobal para extraer las  
marcas de los productos y almacenarlas en el conjunto  
marcasSet.  
  
for (const QString& marca : marcasSet) {  
  
    ui->comboBox_2->addItem(marca);  
  
}  
  
/*
```

1). Limpieza del comboBox_2: ui->comboBox_2->clear();
Limpia el contenido del QComboBox llamado comboBox_2.
Esto asegura que no haya elementos previos antes de
agregar nuevos.

2)Agregar un elemento inicial:
ui->comboBox_2->addItem("Marca");

Agrega un elemento inicial al comboBox_2 con el texto
"Marca".

3)Inicialización de variables: NodoProducto* actualMarca
= catalogoGlobal; QSet<QString> marcasSet;

NodoProducto* actualMarca = catalogoGlobal; permite:

Se inicializa un puntero llamado actualMarca que apunta al inicio de una lista enlazada llamada catalogoGlobal. Esta lista contiene los productos.

QSet<QString> marcasSet; permite:

Se crea un conjunto (QSet) de cadenas de texto (QString). Este conjunto se usará para almacenar las marcas de los productos, asegurando que no haya duplicados (ya que QSet no permite elementos repetidos).

4. Recorrido de la lista enlazada:

actualMarca->dato.marca: Accede al atributo marca del nodo actual.

QString::fromStdString(...): Convierte la cadena estándar de C++ (std::string) a un QString, que es el formato usado por Qt.

marcasSet.insert(...): Inserta la marca en el conjunto. Si la marca ya existe, no se agrega nuevamente (evitando duplicados).

actualMarca = actualMarca->siguiente;: Avanza al siguiente nodo de la lista enlazada. ya creada

5) Agregar las marcas al comboBox_2:

```
for (const QString& marca : marcasSet){  
    ui->comboBox_2->addItem(marca);  
}
```

Itera sobre el conjunto marcasSet y agrega cada marca al comboBox_2. en el for (const QString& marca : marcasSet) se Itera sobre cada elemento del conjunto.

ui->comboBox_2->addItem(marca); luego agrega cada marca como un nuevo elemento en el comboBox_2.

/////////////por lo tanto el proceso seria:

- Limpia el comboBox_2.
- Agrega un elemento inicial ("Marca").
- Recorre una lista enlazada (catalogoGlobal) para extraer las marcas de los productos.
- Usa un QSet para evitar duplicados.
- Agrega las marcas únicas al comboBox_2.

*/

////////////////////////////// aqui
esta la declaracion de la lista de los precios de productos

```
QStringList precios = {"10", "20", "30", "40", "50", "60",  
    "70", "80", "90", "100",  
    "110", "120", "130", "140", "150", "160",  
    "170", "180", "190", "200"};
```

// Declaración de la lista de precios

```
ui->comboBox_3->clear();
```

```
ui->comboBox_3->addItem("Precio min");//Aregar un  
elemento inicial al QComboBox
```

//Aquí se agrega el texto "Precio min" como el primer
elemento del combo box, probablemente como una opción
predeterminada

```
for (const QString& precio : precios) {
```

```
ui->comboBox_3->addItem(precio + "$");//van de 10  
hasta 200 $
```

```
}
```

/*

QStringList: Es una clase de Qt que representa una lista de cadenas de texto (QString). Es similar a un vector o lista en C++ estándar

Inicialización: Se inicializa con una lista de valores entre llaves {}, que son cadenas de texto representando precios en la declaración.

limpieza del ui->comboBox_3->clear(); como:

ui->comboBox_3: Hace referencia a un objeto de tipo QComboBox que forma parte de la interfaz gráfica (UI)

clear(): Este método elimina todos los elementos actualmente presentes en el QComboBox, verifica que esté vacío para luego llenar

cuando agrega elementos al QComboBox:

addItem(): Es un método de QComboBox que agrega un nuevo elemento al desplegable.

cuando hacemos la iteración sobre la lista de precios:

Bucle for: Se utiliza un bucle basado en rango (range-based for loop) para recorrer cada elemento de la lista precios.

const QString& precio: Cada elemento de la lista precios se toma como una referencia constante de tipo QString

`addItem(precio + "$")`: Por cada precio en la lista, se agrega al QComboBox un nuevo elemento que concatena el precio con el símbolo de dólar ("\$").

///////////por lo tanto la fucnionalidad del codigo es que:

Se define una lista de precios (QStringList).

Se limpia el contenido del QComboBox para evitar duplicados.

Se agrega un elemento inicial al QComboBox con el texto "Precio min".

Se recorre la lista de precios y se agregan al QComboBox los precios con el símbolo de dólar (\$) concatenado. ejemplo:
30\$..40\$,...

*/

///////////

```
ui->comboBox_4->clear();
```

```
ui->comboBox_4->addItem("Precio Max");
```

```
for (const QString& precio : precios) {
```

```
    ui->comboBox_4->addItem(precio + "$");
```

```
}
```

```
/*
1. ui->comboBox_4->clear();
```

Limpia el contenido del comboBox_4. de modo que:

ui es un puntero que probablemente apunta a la interfaz gráfica generada por Qt Designer o creada manualmente. comboBox_4 es un objeto de tipo QComboBox, que es un widget desplegable (drop-down list).

El método clear() elimina todos los elementos actualmente presentes en el QComboBox.

```
2. ui->comboBox_4->addItem("Precio Max");
```

Agrega un elemento al comboBox_4 con el texto "Precio Max". De modo que:

El método addItem() añade un nuevo elemento al desplegable.

"Precio Max" es el texto que aparecerá como una opción en el QComboBox.

```
3. for (const QString& precio : precios) { ... }:
```

Itera sobre una lista o colección llamada precios y agrega cada elemento al comboBox_4. De modo que:

precios parece ser una colección (como un QList<QString> o similar) que contiene valores de tipo QString.

const QString& precio es una referencia constante a cada elemento de la colección, lo que evita copias innecesarias y mejora el rendimiento.

El bucle for recorre cada elemento de precios.

4. ui->comboBox_4->addItem(precio + "\$");

Agrega cada precio al comboBox_4, concatenando el símbolo de dólar (\$) al final. De modo que:

precio es un elemento de la colección precios.

precio + "\$" concatena el texto del precio con el símbolo de dólar.

El método addItem() agrega este texto formateado como una nueva opción en el desplegable de opciones.

//////////por lo tanto:

Limpia el contenido del comboBox_4, Agrega un elemento inicial con el texto "Precio Max", Recorre una lista de precios (precios) y agrega cada

precio al comboBox_4, formateado con un símbolo de dólar al final. Nos permite mostrar una lista de precios en un menú desplegable,

permitiendo de tal manera tambien al usuario seleccionar un valor.

*/

}

///////////////////////////////

void

Ventana_Productos::on_categoriaComboBox_currentIndexChanged(const QString &categoria)

{

cargarProductosPorCategoria(categoria);

}

/*1) declaracion de la funcion:

void: Indica que esta función no devuelve ningún valor.

Ventana_Productos::: Esto significa que la función pertenece a la clase Ventana_Productos. Es una función miembro de esta clase.

`on_categoriaComboBox_currentIndexChanged`: Es el nombre de la función. Por convención, en aplicaciones que usan Qt.

`const QString &categoria`: Es un parámetro que recibe la función.

2)Contexto: Qt y señales/slots

En Qt, los widgets (como un QComboBox) emiten señales cuando ocurre un evento, como cambiar la selección.

3)implementacion del cuerpo de la funcion:

`cargarProductosPorCategoria(categoría);`: Aquí se llama a otra función, simultanea con Ventana_Productos. Esta función de cargar los productos

correspondientes a la categoría seleccionada. La categoria
Es el argumento que se pasa a la función
`cargarProductosPorCategoria`.

Contiene el valor actual seleccionado en el QComboBox.

`*/`

/////////////////////////////propósito es cargar productos en una lista gráfica (listWidget_3) a partir de Ventana_Productos

```
void  
Ventana_Productos::cargarProductosPorCategoria(const  
QString& categoriaFiltro)  
{//Estructura del método cargarProductosPorCategoria  
    ui->listWidget_3->clear();
```

```
QMap<QString, QList<Producto>> productosPorCategoria;
```

```
NodoProducto* actual = catalogoGlobal;  
while (actual != nullptr) {  
    QString categoria =  
    QString::fromStdString(actual->dato.categoria);  
  
    if (categoriaFiltro == "Todas las categorías" || categoria  
        == categoriaFiltro) {  
  
        productosPorCategoria[categoria].append(actual->dato);  
    }  
    actual = actual->siguiente;
```

/*

explico:

1)Declaración del método: void
Ventana_Productos::cargarProductosPorCategoria(const
QString& categoriaFiltro)

Este es un método de la clase Ventana_Productos, recibe un parámetro categoriaFiltro de tipo const QString&, que es una cadena de texto utilizada para filtrar los productos por categoría.

2)Limpieza de la lista gráfica: ui->listWidget_3->clear();
Se accede al widget gráfico listWidget_3 (probablemente un QListWidget de Qt) y se limpia su contenido para prepararlo

3)Estructura para almacenar productos por categoría:
 QMap<QString, QList<Producto>> productosPorCategoria;
 Se declara un mapa (QMap) que asocia una categoría (QString) con una lista de productos (QList<Producto>). Esto permite agrupar productos por categoría.

4)> Esto permite agrupar los productos por categoría.

 Recorrido de la lista de productos:

```
NodoProducto* actual = catalogoGlobal;
```

```
    while (actual != nullptr) {
```

Se inicializa un puntero actual que apunta al inicio de una lista enlazada de productos (catalogoGlobal).

Se recorre la lista enlazada usando un bucle while, que continúa hasta que actual sea nullptr (es decir, hasta el final de la lista).

5)Obtención de la categoría del producto actual:

```
QString categoria =  
QString::fromStdString(actual->dato.categoria);
```

Se convierte la categoría del producto actual (actual->dato.categoría) de tipo std::string a QString

6)Filtrado por categoría:

```
if (categoriaFiltro == "Todas las categorías" || categoria ==  
    categoriaFiltro) {
```

```
productosPorCategoria[categoría].append(actual->dato);  
}  
  
//Se verifica si el filtro de categoría es "Todas las categorías"  
// (lo que significa que no se aplica ningún filtro) o si la  
// categoría del producto coincide con el filtro.
```

Si la condición es verdadera, el producto actual (actual->dato) se agrega a la lista correspondiente en el mapa productosPorCategoria. corriendo nodo

7)Avance al siguiente nodo:

```
actual = actual->siguiente;
```

Se mueve el puntero actual al siguiente nodo de la lista enlazada hace el desplazamiento.

///////////por lo tanto su funcionamiento se basa en:

Limpia la lista gráfica.

Recorre una lista enlazada de productos (catalogoGlobal).

Filtrá los productos según la categoría proporcionada (categoriaFiltro).

Agrupa los productos filtrados en un mapa (productosPorCategoria).

Este método es eficiente para manejar listas enlazadas y organizar datos en estructuras.

```
 */  
}
```

```
//////////  
//////////
```

```
for (auto it = productosPorCategoria.begin(); it !=  
    productosPorCategoria.end(); ++it) {  
  
    QString categoria = it.key();  
  
    QList<Producto> productos = it.value();
```

//El resultado es un elemento de lista con un texto personalizado.

```
QListWidgetItem* headerItem = new  
QListWidgetItem("--- " + categoria.toUpper() + " ---");  
  
headerItem->setFlags(headerItem->flags() &  
~Qt::ItemIsSelectable); // No seleccionable  
  
QFont headerFont = headerItem->font();
```

```
    headerFont.setBold(true);  
    headerItem->setFont(headerFont);  
  
    headerItem->setBackground(QBrush(QColor(230, 230,  
                                         230)));  
  
    ui->listWidget_3->addItem(headerItem);
```

/*aqui crea:

1) Creación del objeto QListWidgetItem

2) QListWidgetItem: Es un elemento que se puede
agregar a un QListWidget (un widget de lista en Qt).

new QListWidgetItem(...): Se crea dinámicamente un nuevo
elemento de lista.

categoria.toUpperCase(): Convierte el texto de la variable
categoria a mayúsculas.

"--- " + categoria.toUpperCase() + " ---": Se construye un texto
que incluye la categoría en mayúsculas, rodeada por
guiones (---),formales.

caso de hacer que en Configuración de las propiedades del
elemento Hacer que el elemento no sea seleccionable:

```
headerItem->setFlags(headerItem->flags() &  
                        ~Qt::ItemIsSelectable);
```

`headerItem->flags()`: Obtiene los "flags" actuales del elemento, que son propiedades que definen su comportamiento (por ejemplo, si es seleccionable, editable, etc.).

& `~Qt::ItemIsSelectable`: Se usa una operación bit a bit para desactivar el flag `Qt::ItemIsSelectable`, lo que significa que el elemento no podrá ser seleccionado por el usuario.

`setFlags(...)`: Aplica los nuevos flags al elemento.

b) Cambiar la fuente a negrita: para el encabezado:

`headerItem->font()`: Obtiene la fuente actual del elemento.

`headerFont.setBold(true)`: Modifica la fuente para que sea en negrita.

`headerItem->setFont(headerFont)`: Aplica la fuente modificada al elemento.

c) Cambiar el color de fondo:

`QColor(230, 230, 230)`: Crea un color gris claro utilizando valores RGB (rojo, verde, azul).

`QBrush(...)`: Crea un pincel (QBrush) que se usa para pintar el fondo del elemento.

`setBackground(...)`: Aplica el pincel al fondo del elemento.

3)Agregar el elemento al QListWidget:

ui->listWidget_3: Hace referencia al widget de lista en la interfaz gráfica (probablemente creado en el archivo .ui).

addItem(headerItem): Agrega el elemento headerItem al widget de lista.

Esto coloca el encabezado en la lista visible para el usuario.

*/

/////////////////////////////

/////////bucles: Itera sobre una colección de productos.

```
for (const Producto& p : productos) {
```

```
    QString itemText = QString("%1\n • Marca: %2\n •\n    Precio: $%3\n ")
```

```
        .arg(QString::fromStdString(p.descripcion))
```

```
        .arg(QString::fromStdString(p.marca))
```

```
        .arg(p.precio, 0, 'f', 2)
```

;

```
QListWidgetItem* item = new  
QListWidgetItem(itemText);  
item->setData(Qt::UserRole, p.id);
```

/*1. Bucle for

```
for (const Producto& p : productos) {
```

productos: Es una colección (como un std::vector o std::list) que contiene objetos de tipo Producto.

const Producto& p: Se itera sobre cada elemento de productos. Cada elemento se referencia como p para evitar copias innecesarias y garantizar

que no se modifique el objeto original. Este bucle recorre todos los productos en la colección.

2. Construcción del texto con QString: se construye un texto formateado para representar la información de cada producto.

QString("%1\n • Marca: %2\n • Precio: \$%3\n "): Es una plantilla de texto con marcadores (%1, %2, %3) que serán reemplazados por valores específicos.

.arg(...): Reemplaza los marcadores en orden:

QString::fromStdString(p.descripcion): Convierte la descripción del producto (que está en std::string) a un QString para que sea compatible con Qt.

QString::fromStdString(p.marca): Convierte la marca del producto de std::string a QString.

p.precio, 0, 'f', 2: Formatea el precio como un número flotante con 2 decimales ('f' indica formato de punto flotante).

mstraría: Descripción del producto

- Marca: Marca del producto
 - Precio: \$170.8

3. Creación de un QListWidgetItem:

QListWidgetItem: Es un elemento que se puede agregar a un QListWidget (un widget de lista en Qt).

itemText: El texto formateado creado anteriormente se asigna como el contenido del elemento

4. Asociación de datos adicionales con el elemento:

item->setData(Qt::UserRole, p.id);

setData: Permite asociar datos adicionales al elemento de la lista.

Qt::UserRole: Es una clave personalizada que se usa para almacenar datos específicos del usuario.

p.id: Aquí se asocia el identificador del producto (id) con el elemento de la lista.

*/

//////////////////////////////
//////

```
if (ui->listWidget_3->count() % 2 == 0) {  
    item->setBackground(QBrush(QColor(245, 245,  
        245)));
```

}

```
ui->listWidget_3->addItem(item);
```

/*Condicional if:

La condición `ui->listWidget_3->count() % 2 == 0` verifica si el número de elementos actuales en el `listWidget_3` (un widget de lista, probablemente de la biblioteca Qt) es par. `ui->listWidget_3->count()` obtiene el número de elementos actuales en el widget de lista.

El operador `%` (módulo) calcula el residuo de la división entre `count()` y 2. Si el residuo es 0, significa que el número es par.

`*/`

`}`

```
QListWidgetItem* separator = new QListWidgetItem();
    separator->setFlags(Qt::NoItemFlags);
    separator->setSizeHint(QSize(0, 10));
    ui->listWidget_3->addItem(separator);
    /*
```

Cambio de fondo del elemento:

Si la condición es verdadera (es decir, el número de elementos es par), se ejecuta el bloque dentro del if.

```
item->setBackground(QBrush(QColor(245, 245, 245)));
```

QColor(245, 245, 245) crea un color RGB (gris claro).

QBrush es un objeto que define cómo se pinta el fondo (en este caso, con el color gris claro).

setBackground aplica este pincel como fondo del elemento item

Agregar el elemento a la lista: anexa

Independientemente de si la condición del if se cumple o no, el elemento item se agrega al widget de lista

listWidget_3

*/

}

}

//////////por lo tanto:

/*Si el número de elementos en listWidget_3 es par, el fondo del nuevo elemento (item) se establece en gris claro antes de agregarlo.

Si el número de elementos es impar, el fondo del nuevo elemento no se modifica, pero igual se agrega a la lista

*/

//////////

```
void Ventana_Productos::cargarRecomendaciones()
```

```
{
```

```
    if (!ui->listWidget) {
```

```
        return;
```

```
}
```

```
    ui->listWidget->clear();
```

```
    if (!usuarioActual) {
```

```
        return;
```

```
}
```

Recomendaciones rec =
generarRecomendaciones(usuarioActual); //El resultado se
almacena en la variable rec.

///////////explicacion:

```
/* 1)definicion de funcion: void  
Ventana_Productos::cargarRecomendaciones()
```

void: Indica que la función no devuelve ningún valor.

Ventana_Productos::: Esto indica que la función pertenece a la clase Ventana_Productos. Es una función miembro de esta clase.

cargarRecomendaciones(): Es el nombre de la función, que probablemente se encarga de cargar recomendaciones en la interfaz de usuario.

2) Primera condición: Verificar listWidget:

ui->listWidget: Hace referencia a un widget de tipo lista (probablemente un QListWidget de Qt) que forma parte de la interfaz gráfica.

if (!ui->listWidget): Comprueba si el puntero listWidget es nulo (es decir, si no está inicializado o no existe). Si es nulo, la función termina inmediatamente con return ;

3. Limpiar el contenido del listWidget:

ui->listWidget->clear(): Limpia todos los elementos que puedan estar previamente cargados en el widget de lista.

4. Segunda condición: Verificar usuarioActual: if
(!usuarioActual) { return;}

usuarioActual: Es una variable (probablemente un puntero) que representa al usuario actualmente activo o logueado en el sistema.

if (!usuarioActual): Comprueba si usuarioActual es nulo (es decir, si no hay un usuario activo). Si no hay un usuario, la función termina con return;.

5) genera recomendaciones: Recomendaciones rec = generarRecomendaciones(usuarioActual);

Recomendaciones: Es el tipo de dato de la variable rec. Podría ser una clase, estructura o tipo definido que almacena las recomendaciones generadas.

generarRecomendaciones(usuarioActual): Es una función que toma como parámetro al usuario actual (usuarioActual) y devuelve un conjunto de recomendaciones personalizadas para ese usuario.

///////////////por lo tanto:

Verifica si el listWidget existe: Si no existe, la función termina.

Limpia el contenido del listWidget: Asegura que no haya datos previos.

Verifica si hay un usuario activo: Si no hay usuario, la función termina.

Genera recomendaciones: Llama a una función para obtener recomendaciones personalizadas basadas en el usuario actual

```
*/  
//////////  
////
```

```
auto agregarSeccionRecomendaciones = [&](const  
QString& titulo, ListaProducto lista) {  
  
    if (!lista) {//validacion lista Aquí se verifica si la lista de  
    productos (lista) es nula o vacía. Si lo es, la función termina  
    inmediatamente con return.
```

```
        return;
```

```
}
```

```
// Encabezado de sección  
  
QListWidgetItem* headerItem = new  
QListWidgetItem("★ " + titulo + " ★");  
  
headerItem->setFlags(headerItem->flags() &  
~Qt::ItemIsSelectable);  
  
QFont headerFont = headerItem->font();  
  
headerFont.setBold(true);  
  
headerItem->setFont(headerFont);
```

```
headerItem->setBackground(QBrush(QColor(220, 240,  
255)));
```

ui->listWidget->addItem(headerItem);

/*Declaración de una función lambda:

auto: Permite que el compilador deduzca
automáticamente el tipo de la variable o función.

agregarSeccionRecomendaciones: Es el nombre de la
función lambda.

[&]: Captura todas las variables externas por referencia, lo
que permite que la lambda acceda y modifique variables
definidas fuera de su alcance.

const QString& titulo: Parámetro que recibe un título como
una referencia constante de tipo QString.

ListaProducto lista: Segundo parámetro que parece ser una
lista de productos (probablemente una estructura o clase
definida en el programa).

Creación del encabezado de sección: QListWidgetItem*
headerItem = new QListWidgetItem("★ " + titulo + " ★");

QListWidgetItem: Es un elemento de lista que se puede
agregar a un QListWidget (un widget de lista en Qt).

new QListWidgetItem: Se crea dinámicamente un nuevo
elemento de lista.

"★ " + titulo + " ★" y el título proporcionado como parámetro.

configuración de encabezado:

setFlags: Configura las propiedades del elemento.

headerItem->flags(): Obtiene las banderas actuales del elemento.

& ~Qt::ItemIsSelectable: Desactiva la capacidad de seleccionar este elemento en la lista.

headerItem->font(): Obtiene la fuente actual del elemento.

headerFont.setBold(true): Configura la fuente para que sea negrita.

headerItem->setFont(headerFont): Aplica la fuente modificada al elemento.

QBrush: Define un pincel para pintar el fondo del elemento.

QColor(220, 240, 255): Especifica un color de fondo en formato RGB (un azul claro).

para agregar encabezado de la lista:

ui->listWidget: Hace referencia al widget de lista en la interfaz de usuario

addItem(headerItem): Agrega el encabezado configurado como un nuevo elemento en el widget de lista

```
*/  
//////////  
// Agregar productos (máximo 10)  
int contador = 0;  
NodoProducto* actual = lista;  
while (actual != nullptr && contador < 10) {  
    Producto p = actual->dato;  
    QString itemText = QString("%1\n • Marca: %2\n •  
    Precio: $%3\n • Calidad: %4/5")//Formateo del texto  
    .arg(QString::fromStdString(p.descripcion))  
    .arg(QString::fromStdString(p.marca))  
    .arg(p.precio, 0, 'f', 2)  
    .arg(p.calidad);  
  
    QListWidgetItem* item = new  
    QListWidgetItem(itemText);  
    item->setData(Qt::UserRole, p.id);  
  
    ui->listWidget->addItem(item);  
    contador++;
```

```
actual = actual->siguiente;
```

```
}
```

```
/*
```

1)declaracion de variables iniciales:

contador: Se inicializa en 0 y sirve para limitar el número de iteraciones del bucle (máximo 10 elementos).

actual: Es un puntero que apunta al primer nodo de la lista enlazada (lista). Este nodo contiene un objeto Producto y un puntero al siguiente nodo.

2)bucle mientras sea :

actual != nullptr: Asegura que no se ha llegado al final de la lista enlazada.

contador < 10: Limita el número de iteraciones a 10, para no agregar más de 10 elementos al QListWidget.

3) Formateo del texto:

```
QString itemText = QString("%1\n • Marca: %2\n • Precio:  
$%3\n • Calidad: %4/5")
```

```
.arg(QString::fromStdString(p.descripcion))
```

```
.arg(QString::fromStdString(p.marca))
```

```
.arg(p.precio, 0, 'f', 2)
```

```
.arg(p.calidad);
```

Se crea un QString llamado itemText que contiene la información del producto formateada para mostrarla en el QListWidget.

QString::arg: Inserta valores en los marcadores %1, %2, etc.

QString::fromStdString: Convierte cadenas estándar de C++ (std::string) a QString, que es el formato usado por Qt.

p.precio, 0, 'f', 2: Formatea el precio como un número flotante con 2 decimales.

p.calidad: Representa la calidad del producto (por ejemplo, de 1 a 5). como en las app de playstore, o calificacionde productos digitales y catalogos

4)elementos :

QListWidgetItem: Se crea un nuevo elemento de lista con el texto formateado (itemText).

setData(Qt::UserRole, p.id): Asocia un dato adicional al elemento (en este caso, el id del producto).

5)agregar elemento: ui->listWidget->addItem(item);

6)Avanzar al siguiente nodo y actualizar el contador:

contador++;

actual = actual->siguiente;

//////////7el flujo del programa seria:

Se inicializan las variables.

Se recorre la lista enlazada nodo por nodo, hasta que:

Se procesen 10 elementos, o

Se llegue al final de la lista.

En cada iteración:

Se extraen los datos del nodo actual.

Se formatea un texto descriptivo del producto.

Se crea un elemento de lista (QListWidgetItem) con el texto
y un dato adicional (id).

Se agrega el elemento al QListWidget.

Se avanza al siguiente nodo.

*/

//////////7

// Separador

```
QListWidgetItem* separator = new QListWidgetItem();  
separator->setFlags(Qt::NoItemFlags);  
separator->setSizeHint(QSize(0, 15));
```

```
ui->listWidget->addItem(separator);//addItem(separator):  
    Inserta el separador en el QListWidget  
};
```

//Se declara un puntero llamado separator que apunta a un nuevo objeto de tipo QListWidgetItem.

//un elemento en el QListWidget, pero que será usado como un separador visual.

// Configuración de las propiedades del separador
/*

setFlags(Qt::NoItemFlags): Desactiva todas las interacciones del usuario con este elemento.

Esto significa que el separador no será seleccionable, clickeable ni editable.

setSizeHint(QSize(0, 15)): Define un tamaño sugerido para el elemento. En este caso, se establece una altura de 15 píxeles

y un ancho de 0 (el ancho será ajustado automáticamente por el QListWidget).

*/

//////////////////////////////
//

```
agregarSeccionRecomendaciones("MARCAS PREFERIDAS",
    rec.porMarcasPreferidas);

agregarSeccionRecomendaciones("OTRAS MARCAS
    FRECUENTES", rec.porOtrasMarcasFrecuentes);

agregarSeccionRecomendaciones("CATEGORÍA
    FRECUENTE", rec.porCategoriaFrecuente);

agregarSeccionRecomendaciones("CALIDAD SIMILAR",
    rec.porCalidad);

/*
```

La función agregarSeccionRecomendaciones probablemente
está diseñada para agregar una sección de
recomendaciones a algún
tipo de estructura de datos o interfaz

Primer parámetro: El primer argumento es una cadena de
texto (por ejemplo, "MARCAS PREFERIDAS")

Segundo parámetro: El segundo argumento parece ser una
variable

rec.porMarcasPreferidas: Datos relacionados con marcas
preferidas.

rec.porOtrasMarcasFrecuentes: Datos sobre otras marcas
frecuentes.

rec.porCategoriaFrecuente: Información sobre categorías
frecuentes.

rec.porCalidad: Información sobre calidad similar.

Esto sugiere que rec es una instancia de una clase o estructura que agrupa toda la información de recomendaciones.

*/

//////////////////////////////

if (ui->listWidget->count() == 0) {

/*condicional if:

ui->listWidget: Hace referencia a un widget de tipo QListWidget que probablemente está definido en la interfaz gráfica (UI) de la aplicación.

count(): Es un método de QListWidget que devuelve el número de elementos actualmente en la lista.

== 0: Comprueba si la lista está vacía. Si no hay elementos en el listWidget, se ejecuta el bloque de código dentro del if.

*/

QListWidgetItem* item = new QListWidgetItem("No hay recomendaciones disponibles");

/*crea nuevo eleemnto:

QListWidgetItem: Es un objeto que representa un elemento dentro de un QListWidget.

new QListWidgetItem("No hay recomendaciones disponibles"); Crea dinámicamente un nuevo elemento con el texto "No hay recomendaciones disponibles".

**/

item->setFlags(item->flags() & ~Qt::ItemIsSelectable);
ui->listWidget->addItem(item); //addItem(item): Añade el nuevo elemento (item) al listWidget.

}

}

/*Este código verifica si el listWidget está vacío. Si no hay elementos, crea un nuevo elemento con el texto "No hay recomendaciones disponibles",

modificación de propiedades de elementos

item->flags(): Obtiene las propiedades actuales (flags) del elemento, como si es seleccionable, editable, etc.

& ~Qt::ItemIsSelectable: Utiliza una operación bit a bit para desactivar el flag Qt::ItemIsSelectable, lo que significa que el elemento no será seleccionable por el usuario.

~Qt::ItemIsSelectable: Es el complemento bit a bit del flag Qt::ItemIsSelectable.

&: Combina los flags actuales con el complemento, desactivando únicamente la propiedad de seleccionabilidad.

```
*/  
///////////Función: mostrarDetallesProducto  
  
void Ventana_Productos::mostrarDetallesProducto(int id)  
{//Esta función pertenece a la clase Ventana_Productos y  
tiene como propósito mostrar los detalles de un producto  
específico, identificado por su id.  
  
    NodoProducto* actual = catalogoGlobal;  
    while (actual != nullptr) {  
        if (actual->dato.id == id) {  
            /*  
             aquí se comparan los id:  
             actual->dato: Accede al dato almacenado en el nodo  
actual. Este dato parece ser un objeto o estructura que  
contiene información del producto.  
actual->dato.id: Accede al campo id del producto  
almacenado en el nodo actual.  
actual->dato.id == id: Compara el id del producto actual con  
el id proporcionado como parámetro. Si coinciden, significa  
que se encontró el producto deseado
```

*/

productodialog dialog(actual->dato,
this);//actual->dato: Los datos del producto
encontrado./this: Un puntero a la instancia actual de
Ventana_Productos,
dialog.exec();//dialog.exec(): Muestra el diálogo de
manera modal

/*void

Ventana_Productos::mostrarDetallesProducto(int id)

void: Indica que la función no devuelve ningún valor.

Ventana_Productos::: Especifica que esta función pertenece
a la clase Ventana_Productos.

mostrarDetallesProducto(int id): Es el nombre de la función,
y recibe un parámetro entero id, que representa el
identificador del producto que se desea buscar

variables creadas:

NodoProducto* actual: Declara un puntero llamado actual
que apunta a un nodo de tipo NodoProducto.

catalogoGlobal: Es una variable (probablemente un puntero)
que apunta al inicio de la lista enlazada que contiene todos
los productos disponibles

```
/*
// Registrar en el historial
if (usuarioActual) {
    insertarLista(usuarioActual->historial,
        std::to_string(id));
}

// Actualizar recomendaciones inmediatamente
cargarRecomendaciones(); //La función
cargarRecomendaciones() se llama para realizar esta
actualización.

}
break;
}

actual = actual->siguiente;
}

/*Condicional if (usuarioActual):
```

Este bloque verifica si el puntero usuarioActual no es nulo. Es decir, se asegura de que existe un usuario actual antes de intentar

realizar operaciones con él.

Si usuarioActual es válido (no es nullptr), se ejecuta el bloque de código dentro de las llaves {}.

Llamada a insertarLista:

Dentro del bloque, se llama a la función insertarLista con dos argumentos:

usuarioActual->historial: Esto parece ser un atributo del objeto apuntado por usuarioActual, probablemente una lista

o estructura que almacena un historial de datos.

std::to_string(id): Convierte el valor de id (probablemente un entero) a una cadena de texto (std::string) para poder insertarlo en el historial.

otros aspectos del código:

Lista enlazada: La estructura actual = actual->siguiente es típica en listas enlazadas, donde cada nodo tiene un puntero al siguiente.

Funciones externas: insertarLista y cargarRecomendaciones son funciones definidas en otro lugar del programa.

```
 */
}

///////////
////

void Ventana_Productos::on_pushButton_clicked()
{ /*declaracion dde la funcion:
```

void: Es el tipo de retorno de la función, lo que significa que esta función no devuelve ningún valor.

Ventana_Productos::: Indica que esta función pertenece a la clase Ventana_Productos. Es una función miembro de esta clase.

on_pushButton_clicked: Es el nombre de la función. Por convención en Qt, este tipo de nombres se utilizan para manejar eventos,

en este caso, el evento de clic en un botón llamado pushButton.

```
*/
```

```
QString textoBusqueda = ui->lineEdit->text();
QString categoria = ui->comboBox->currentText();
QString marca = ui->comboBox_2->currentText();
```

```
QString precioMinStr = ui->comboBox_3->currentText();  
QString precioMaxStr = ui->comboBox_4->currentText();
```

`/*void:` Es el tipo de retorno de la función, lo que significa que esta función no devuelve ningún valor.

`Ventana_Productos:::` Indica que esta función pertenece a la clase Ventana_Productos. Es una función miembro de esta clase.

`on_pushButton_clicked:` Es el nombre de la función. Por convención en Qt, este tipo de nombres se utilizan para manejar eventos,

en este caso, el evento de clic en un botón llamado `pushButton`.

`///////////////////////////////`importante: VARIABLES LOCALES CON EL ENTORNO QT:

2. Variables locales

Dentro de la función, se declaran varias variables locales que capturan datos de la interfaz gráfica (UI):

a) `QString textoBusqueda`

```
QString textoBusqueda = ui->lineEdit->text();
```

QString: Es una clase de Qt que representa cadenas de texto.

ui->lineEdit: Hace referencia a un widget de entrada de texto (un campo de texto) llamado lineEdit en la interfaz gráfica.

.text(): Es un método que obtiene el texto ingresado por el usuario en el lineEdit.

textoBusqueda: Almacena el texto ingresado por el USUARIO.

b) QString categoria

```
QString categoria = ui->comboBox->currentText();
```

ui->comboBox: Hace referencia a un widget desplegable (un QComboBox) llamado comboBox.

.currentText(): Obtiene el texto actualmente seleccionado en el comboBox.

categoria: Almacena la categoría seleccionada por el usuario

c) QString marca

```
QString marca = ui->comboBox_2->currentText();
```

Similar al caso anterior, pero hace referencia a otro QComboBox llamado comboBox_2.

marca: Almacena la marca seleccionada por el usuario.

d) QString precioMinStr y QString precioMaxStr

precioMinStr: Almacena el texto seleccionado en el comboBox_3, que probablemente representa el precio mínimo.

precioMaxStr: Almacena el texto seleccionado en el comboBox_4, que probablemente representa el precio máximo.

Esta función estar diseñada para capturar los valores ingresados o seleccionados por el usuario en varios widgets de la interfaz gráfica

(un campo de texto y varios menús desplegables). Estos valores probablemente se usarán para realizar una búsqueda o filtrar productos según los criterios especificados.

4. Contexto Qt:

ui: Es un puntero a la interfaz gráfica generada por Qt Designer. Permite acceder a los widgets definidos en el archivo .ui.

lineEdit, comboBox, comboBox_2, etc.: Son los nombres de los widgets definidos en el archivo .ui.

*/

//////////////////////////////
//7

```
double precioMin = 0.0;  
double precioMax = std::numeric_limits<double>::max();
```

```
QRegularExpression re("(\\d+\\.?\\d*)");
```

```
QRegularExpressionMatch matchMin =  
    re.match(precioMinStr);
```

```
QRegularExpressionMatch matchMax =  
    re.match(precioMaxStr);
```

//re.match(precioMinStr): Intenta encontrar una coincidencia en la cadena precioMinStr

//QRegularExpressionMatch: Es una clase que almacena el resultado de la coincidencia.

/*precioMin: Se inicializa con el valor 0.0, lo que indica el límite inferior del rango de precios.

precioMax: Se inicializa con el valor máximo posible para un tipo de dato double. Esto se logra utilizando std::numeric_limits<double>::max(), que pertenece a la biblioteca estándar <limits>. Sirve para establecer un límite superior

```
*/  
//////////  
//7  
if (matchMin.hasMatch()) {  
    precioMin = matchMin.captured(1).toDouble();  
}  
  
if (matchMax.hasMatch()) {  
    precioMax = matchMax.captured(1).toDouble();
```

/*Métodos hasMatch() y captured(1): Determinan si hay coincidencias y extraen datos específicos.

Conversión a double: Convierte los valores capturados a un formato numérico

*/

}

/* en contexto:

f (matchMin.hasMatch())

Comprueba si el objeto matchMin contiene una coincidencia válida.

hasMatch(): Es probablemente un método de una clase relacionada con expresiones regulares (como QRegularExpressionMatch en Qt).

Este método devuelve true si se encontró una coincidencia en el texto analizado.

precioMin = matchMin.captured(1).toDouble();

Si matchMin tiene una coincidencia, se extrae un valor capturado (probablemente un número) y se convierte a un tipo double.

`captured(1)`: Obtiene el contenido del primer grupo capturado en la expresión regular.

`toDouble()`: Convierte el texto capturado en un número de tipo double.

Resultado: es El valor convertido se asigna a la variable precioMin.

```
if (matchMax.hasMatch())
```

Similar al caso anterior, verifica si matchMax contiene una coincidencia válida.

```
precioMax = matchMax.captured(1).toDouble();
```

Si matchMax tiene una coincidencia, se extrae el primer grupo capturado, se convierte a double y se asigna a la variable precioMax.

```
*/
```

```
//////////////////////////////  
|||||||
```

```
if (precioMin > precioMax) {
```

```
QMessageBox::warning(this, "Error", "El precio mínimo  
no puede ser mayor al precio máximo");  
  
return;  
}
```

ListaProducto resultados = catalogoGlobal;

/*de aqui dependen los parametros de producto:

Condicional if:

La condición if (precioMin > precioMax) verifica si el valor de precioMin (precio mínimo) es mayor que el de precioMax (precio máximo).

Si esta condición es verdadera, significa que hay un error lógico en los valores ingresados, ya que el precio mínimo no debería ser mayor al precio máximo.

QMessageBox::warning:

Esta línea muestra un cuadro de diálogo de advertencia al usuario. Es parte de la biblioteca Qt, que se utiliza para desarrollar interfaces gráficas en C++.

Parámetros:

this: Hace referencia al objeto actual (probablemente una ventana o formulario).

"Error": Es el título del cuadro de diálogo. sin cerrar "El precio mínimo no puede ser mayor al precio máximo": Es el mensaje que se muestra al usuario para explicar el problema y vuelva a generar

////////7 del catalogo de los productos:

precioMin y precioMax.

ListaProducto resultados = catalogoGlobal; // fragmento
de codigo.

Esta línea se ejecuta solo si la condición del if no se cumple (es decir, si precioMin no es mayor que precioMax).

Aquí se está inicializando una variable llamada resultados de tipo ListaProducto (probablemente una clase o estructura definida en el programa).

Se le asigna el valor de catalogoGlobal, que parece ser una lista o colección global de productos.

```
/*
///////////
//utiliza condicionales if para realizar ciertas operaciones
dependiendo de las condiciones evaluadas.

if (!textoBusqueda.isEmpty()) {

    resultados = buscarPorDescripcion(resultados,
        textoBusqueda.toStdString());

}

/*
```

PRIMERA CONDICION:

`!textoBusqueda.isEmpty()` verifica si el objeto `textoBusqueda` no está vacío.

El operador `!` (negación lógica) invierte el resultado de `isEmpty()`. Si `isEmpty()` devuelve `false` (es decir, el texto no está vacío), la condición se cumple.

Si la condición es verdadera, se llama a la función
buscarPorDescripcion().

Esta función toma dos parámetros:
resultados: una lista, vector o colección que contiene datos.

textoBusqueda.toStdString(): convierte el objeto
textoBusqueda (posiblemente un QString) a un std::string
para que sea compatible con la función.

Resultado: El resultado de buscarPorDescripcion() se asigna
nuevamente a la variable resultados, actualizándola con los
datos filtrados según la descripción

SEGUNDA CONDICION:

Aquí se evalúan dos condiciones combinadas con el
operador lógico && (AND):

categoria != "Categorias": verifica que el valor de categoria
no sea igual a la cadena "Categorias".

Esto podría ser un valor predeterminado o genérico que no
se desea procesar.

!categoria.isEmpty(): verifica que categoria no esté vacía,
similar a la primera condición.

Ambas condiciones deben ser verdaderas para que el
bloque de código dentro del if se ejecute.

Si ambas condiciones son verdaderas, se llama a la función buscarPorCategoria().

Esta función también toma dos parámetros:

resultados: la colección actual de datos.

categoria.toStdString(): convierte categoria (posiblemente un QString) a un std::string.

Resultado: El resultado de buscarPorCategoria() se asigna nuevamente a resultados, actualizándolo con los datos filtrados según la categoría.

*/

```
if (categoria != "Categorias" && !categoria.isEmpty()) {
```

```
    resultados = buscarPorCategoria(resultados,  
                                     categoria.toStdString());
```

```
}
```

```
/*implementacion:
```

Primera parte: Si el texto de búsqueda no está vacío, se filtran los resultados por descripción.

Segunda parte: Si la categoría no es "Categorias" y no está vacía, se filtran los resultados por categoría.

Actualización de resultados: En ambos casos, la variable resultados se actualiza con los datos filtrados.

///operadores y metodos usados:

Operadores lógicos:

! (NOT): Invierte el valor lógico.

&& (AND): Requiere que ambas condiciones sean verdaderas.

Métodos de cadenas:

isEmpty(): Comprueba si una cadena está vacía.

toStdString(): Convierte un QString a un std::string.

*/

//////////////////////////////7

```
if (marca != "Marca" && !marca.isEmpty()) {  
    resultados = buscarPorMarca(resultados,  
        marca.toStdString());  
}
```

```
resultados = buscarPorRangoPrecios(resultados,  
    precioMin, precioMax);
```

/*

Condicional if:

marca != "Marca": Se verifica que la variable marca no sea igual a la cadena "Marca". Esto podría ser un valor predeterminado o

un marcador que indica que no se ha seleccionado una marca válida.

&& (AND lógico): Ambas condiciones deben cumplirse para que el bloque de código dentro del if se ejecute.

!marca.isEmpty(): Se verifica que la variable marca no esté vacía. Esto asegura que haya un valor válido en la variable antes de proceder.

Propósito: Este if asegura que solo se realice la búsqueda por marca si la marca es válida (no es "Marca" y no está vacía

/////////

Llamada a buscarPorMarca:

Si la condición del if es verdadera, se ejecuta esta línea:

```
resultados = buscarPorMarca(resultados,  
                           marca.toStdString());
```

buscarPorMarca: Es una función que probablemente filtra los resultados existentes (resultados) según la marca proporcionada.

marca.toStdString(): Convierte la variable marca (que parece ser un objeto de tipo QString, común en Qt) a un std::string, que es el tipo de cadena estándar en C++.

///

Búsqueda por rango de precios:

Independientemente de si se ejecutó el bloque del if o no, esta línea se ejecuta:

```
resultados = buscarPorRangoPrecios(resultados, precioMin,  
precioMax);
```

buscarPorRangoPrecios: Es otra función que filtra los resultados existentes (resultados) según un rango de precios definido por precioMin (precio mínimo) y precioMax (precio máximo).

Propósito: Refinar los resultados para que solo incluyan elementos dentro del rango de precios especificado.

/////////////// aplicar filtros:

Si la marca es válida (no es "Marca" y no está vacía), se filtran los resultados por marca.

Luego, los resultados (ya filtrados o no) se refinan aún más aplicando un filtro por rango de precios.

```
/*
///////////
/////////
ui->listWidget_2->clear();

if (resultados == nullptr) {
    QListWidgetItem* item = new QListWidgetItem("No se
        encontraron productos");
    ui->listWidget_2->addItem(item);
    return;
}

/*limpieza del ui->listWidget_2->clear();
ui->listWidget_2: Hace referencia a un widget de tipo
QListWidget que forma parte de la interfaz gráfica de
usuario (UI) creada con Qt.

clear(): Este método elimina todos los elementos que
actualmente están en el listWidget_2, dejando la lista vacía.
```

/verificacion de un puntero:

resultados: Es un puntero que se está verificando para saber si apunta a algo válido o si es nullptr (es decir, no apunta a ningún objeto).

== nullptr: Comprueba si el puntero no tiene un valor asignado. Si es nullptr, significa que no hay resultados disponibles

//creacion de nuevo elemento en la lista:

QListWidgetItem: Es un objeto que representa un elemento individual dentro de un QListWidget.

new QListWidgetItem("No se encontraron productos"): Se crea dinámicamente un nuevo elemento con el texto "No se encontraron productos".

///agregar eleemnto:ui->listWidget_2->addItem(item);

addItem(item): Este método agrega el elemento recién creado (item) al listWidget_2, para que sea visible en la interfaz gráfica

*/

//////////
//////////

```

NodoProducto* actual = resultados;

    while (actual != nullptr) {

        Producto p = actual->dato;

        QString itemText = QString::fromStdString(
            p.descripcion + " - " + p.marca + " - $" +
            std::to_string(p.precio)
        );

        QListWidgetItem* item = new
        QListWidgetItem(itemText);

        item->setData(Qt::UserRole, p.id);

        ui->listWidget_2->addItem(item);

        actual = actual->siguiente;

    }

}

```

/*Declaración de un puntero a un nodo de la lista enlazada:

```

NodoProducto* actual = resultados;

```

Aquí, actual es un puntero que apunta al inicio de una lista enlazada de nodos (NodoProducto).

resultados es el nodo inicial (o cabeza) de la lista enlazada.

Este bucle recorre la lista enlazada. Se ejecuta mientras actual no sea nullptr, lo que indica que aún hay nodos por procesar

Acceso al dato del nodo actual: Producto p = actual->dato;

Cada nodo (NodoProducto) contiene un objeto Producto almacenado en su atributo dato.

Aquí, se extrae el objeto Producto del nodo actual y se almacena en la variable p

-Construcción de un texto descriptivo: porque:

```
QString itemText = QString::fromStdString(  
    p.descripcion + " - " + p.marca + " - $" +  
    std::to_string(p.precio)  
);
```

explico:

Se crea un texto descriptivo para el producto combinando su descripción (p.descripcion), marca (p.marca) y precio (p.precio).

std::to_string(p.precio) convierte el precio (un número) a una cadena.

QString::fromStdString convierte la cadena estándar de C++ (std::string) a un QString, que es el tipo de cadena usado en Qt

Creación de un elemento de lista (QListWidgetItem):

```
QListWidgetItem* item = new QListWidgetItem(itemText);
```

Se crea un nuevo elemento de lista (QListWidgetItem) con el texto descriptivo generado (itemText) mientras recorre

Asociación de datos adicionales al elemento: recopilar información para identificar el producto seleccionado)

```
item->setData(Qt::UserRole, p.id);
```

Se asocia un dato adicional al elemento de lista usando setData. En este caso, se guarda el identificador del producto (p.id) en el rol de usuario (Qt::UserRole).

```
ui->listWidget_2->addItem(item);    aqui el elemento de lista (item) se añade a un widget de lista (listWidget_2) que forma parte de la interfaz gráfica.
```

//////////7por lo tanto esta parte realiza:

Se inicializa el puntero actual con el nodo inicial de la lista.

Se recorre la lista enlazada nodo por nodo.

Para cada nodo:

Se extrae el objeto Producto.

Se genera un texto descriptivo.

Se crea un elemento de lista con ese texto.

Se asocia información adicional (el ID del producto).

Se añade el elemento al widget de lista.

El puntero avanza al siguiente nodo hasta que no queden más nodos.

*/

///mpl
ementación de un destructor en C++ para una clase llamada
Ventana_Productos.

Ventana_Productos::~Ventana_Productos()

{

 delete ui;

}

/*Ventana_Productos::~Ventana_Productos()

Clase y Destructor en qt

Ventana_Productos es el nombre de la clase.

~Ventana_Productos() es el destructor de la clase. En C++, los destructores son funciones especiales que se ejecutan automáticamente cuando un objeto de la clase es destruido (por ejemplo, cuando sale de alcance o se elimina manualmente con delete).

El prefijo ~ indica que es un destructor.

Propósito del Destructor:

Su función principal es liberar recursos asignados dinámicamente (como memoria, archivos abiertos, etc.) para evitar fugas de memoria.

2. delete ui;

ui:Es un puntero (probablemente a un objeto de interfaz de usuario, como un formulario o ventana gráfica) que fue creado dinámicamente con new en algún momento durante la vida del objeto Ventana_Productos.

delete:La palabra clave delete se utiliza para liberar la memoria asignada dinámicamente con new. Esto asegura que no haya fugas de memoria.

*/

///////////////////////////////

void Ventana_Productos::on_Volver_clicked()

```
{  
    this->close();  
  
    MainWindow *ventanaP = new MainWindow();  
  
    ventanaP->show();  
  
}  
  
/*Explicación:
```

Declaración de la función miembro:
void
Ventana_Productos::on_Volver_clicked():

Es una función miembro de la clase Ventana_Productos.

El prefijo Ventana_Productos:: indica que esta función pertenece a la clase Ventana_Productos.

El nombre on_Volver_clicked sugiere que esta función se ejecuta cuando se hace clic en un botón llamado "Volver" (probablemente conectado a una señal en un framework como Qt).

El tipo de retorno es void, lo que significa que esta función no devuelve ningún valor.

Cierre de la ventana actual:

this->close();: this es un puntero que hace referencia a la instancia actual de la clase Ventana_Productos.

close() es un método que probablemente pertenece a la clase base (como QWidget en Qt) y se utiliza para cerrar la ventana actual.

Creación de una nueva ventana:

MainWindow *ventanaP = new MainWindow();: Aquí se crea dinámicamente una nueva instancia de la clase MainWindow utilizando el operador new.

ventanaP es un puntero que apunta a esta nueva instancia.

La clase MainWindow parece ser otra ventana o interfaz gráfica que se mostrará después de cerrar la ventana actual.

/////////7visualizacion

Mostrar la nueva ventana:ventanaP->show();:

Se llama al método show() de la instancia ventanaP para mostrar la nueva ventana en pantalla.

Este método es común en frameworks de interfaces gráficas como Qt y se utiliza para hacer visible una ventana.

*/

//////////////////////////////

