**UNIVERSIDAD DE GUANAJUATO**

DIVISIÓN DE INGENIERÍAS CAMPUS IRAPUATO-SALAMANCA

LIC. EN ING. EN SISTEMAS COMPUTACIONALES

ALGORITMOS Y ESTRUCTURAS DE DATOS

PROFESOR: DR. CARLOS HUGO GARCÍA CAPULÍN

**NO. DE TAREA:** 06

**NOMBRE DE LA TAREA:**

CREACIÓN DINÁMICA DE ESTRUCTURAS Y SU MANEJO MEDIANTE APUNTADORES

**ESTUDIANTE:**

MANRÍQUEZ COBIÁN ROGELIO

**FECHA DE ENTREGA:**

18 DE SEPTIEMBRE DEL 2020



**Problema**

Implementar un programa que realice las operaciones aritméticas básicas (+, -, \*, /) de dos números complejos, utilizando estructuras de datos y funciones, apuntadores y memoria dinámica.

Para recordar qué es una estructura de datos, se deja la siguiente explicación:

ESTRUCTURAS DE DATOS

Una estructura es una agrupación de datos de diferente tipo que tienen un nombre común.

SINTAXIS para la declaración de la estructura:

typedef struct {

tipo\_de\_dato Nombre1;

tipo\_de\_dato Nombre2;

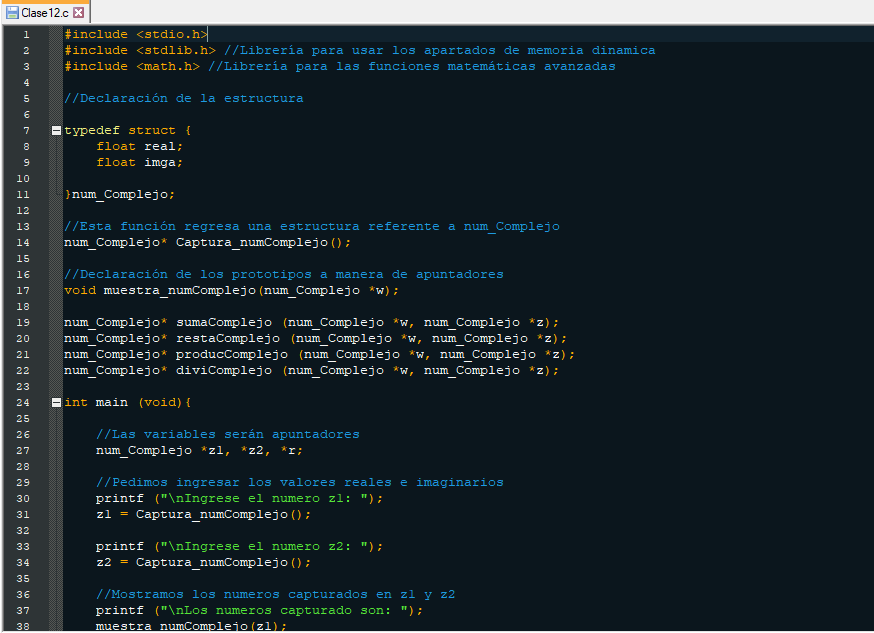
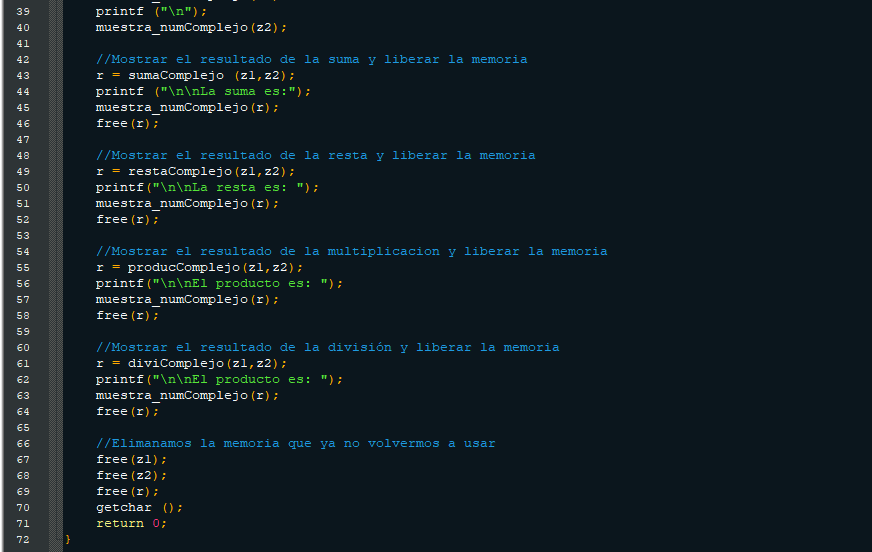
...

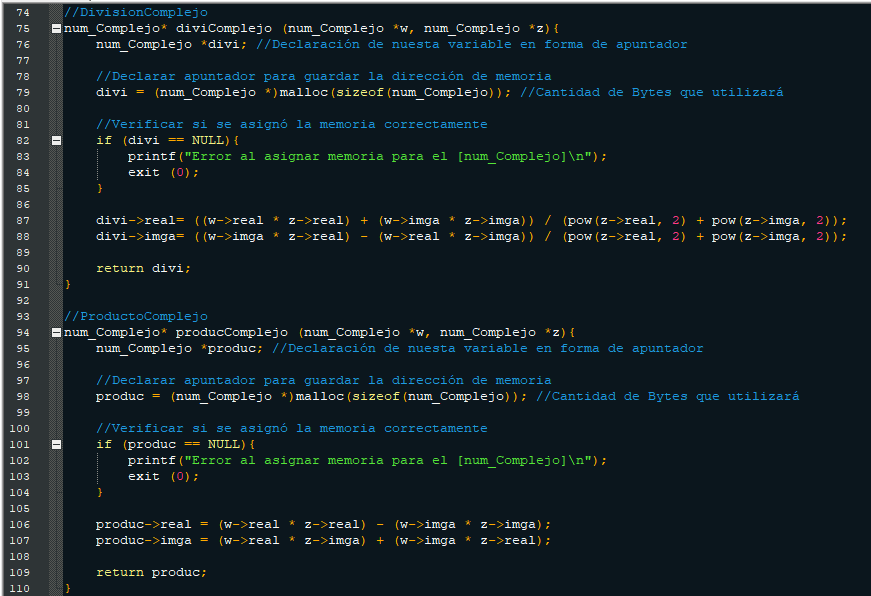
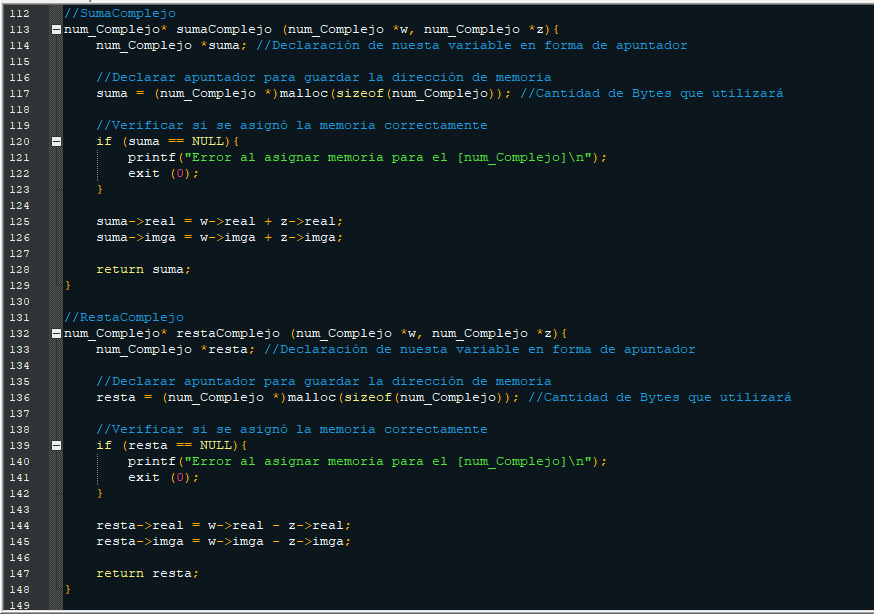
tipo\_de\_dato NombreN;

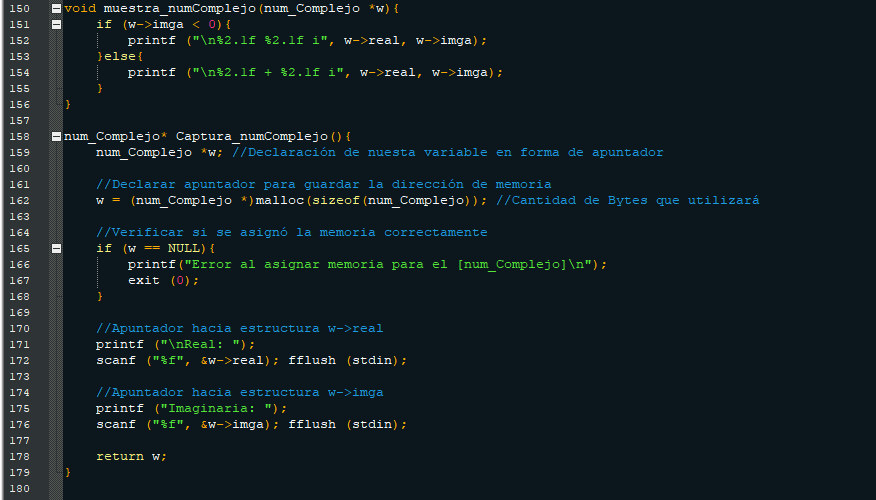
}Nombre\_De\_La\_Estrucura;

Nombre\_De\_La\_Estrucura variable1;

**Solución Implementada:**







**Pruebas y Resultados:**

Ahora que ya tenemos el problema planteado hay que implementar una solución de acuerdo con lo que se nos pide resolver en el reporte.

Iniciamos creando un nuevo archivo en nuestro editor de textos Notepad ++ el cual lo guardaremos con el nombre de T06.c (“En mi caso tiene el nombre de Clase12, ya que este ejercicio se planteó en la clase).

Comenzamos escribiendo nuestro “**#include <stdio.h>**”, “**#include <math.h>**” y “**#include <stdlib.h>**”; la librería “**math**” sirve principalmente para operaciones matemáticas más avanzados como raíz cuadrada, elevar un número a la “n” potencia, etc. La librería “**stdlib**”, nos servirá en la sección para poder declarar las variables en tipo de apuntador y asignarle memoria dinámica.

Ahora, lo interesante será que después de escribir nuestras librerías de preprocesador tendremos que crear nuestra estructura de datos, el cual está declarado con la nomenclatura “**typedef struct**” para evitarnos una cosa tediosa en el código “**main**”.

Dentro de nuestra estructura declararemos dos variables de tipo “**float**”, el primero tendrá como nombre “**real**” y el otro tendrá como nombre “**imga**”, por último, declaramos nuestra estructura con el nombre de “**numero\_Complejo**”. Entonces, escribiremos nuestra función “**main**”, donde declararemos nuestra estructura de datos con el nombre de tres variables que estaremos usando para las funciones y operaciones dentro de estas operaciones de números complejos, estas variables serán z1, z2, r.

Para la primer parte de nuestro código haremos el usuario ingrese los números para guardarlos en la variable “**z1** y **z2**” en el cual se mandará a una función “**Captura\_numComplejo**” que esta función misma se declarará dentro una variable “**num\_Complejo**” en “**w**”, donde guardemos los números reales e imaginarios dentro de la variable “**w**” que está apuntando a la estructura “**num\_Complejo**” y por último retornamos la dirección de “**w**”.

NOTA: Declarar la función **num\_Complejo**\* **Captura\_numComplejo ()** como prototipo.

Ahora, lo que hacemos es mostrar los números capturados por la función, en otra función vacía llamada “**void muestra\_numComplejo (num\_Complejo \*w)**” en la cual tenemos almacenado los valores gracias al apuntador “**w**”; ahora solo dentro de esta función tendremos una condición para la poder imprimir en pantalla los datos capturados, la condición es de que el apuntador “**w**” dirigido a la variable “**imga**” de la estructura es menor a cero, imprima la parte positiva (real) y la parte negativa (imaginaria), sino, imprime ambos números complejos de manera positiva.

Esta función solo devolverá los valores a la función “**main**” para mostrar la dirección en memoria que almacena “**z1** y **z2**”.

NOTA: Declarar la función **void muestra\_numComplejo (num\_Complejo \*w)** como prototipo.

Entonces, llega lo más importante del problema, implementar una solución en las operaciones aritméticas.

Por primera instancia, resolveremos, la operación “**suma**”.

Esta función tendrá como valor un “**num\_Complejo**\* **sumaComplejo (num\_Complejo \*w, num\_Complejo \*z)**” dentro como argumento de nuestra función recibirá dos parámetros a manera de apuntadores para poder almacenar los datos que realicemos con ellos de manera dinámica.

Dentro de nuestra función declaremos una variable de tipo “**num\_Complejo**” con el nombre “**suma**” a manera de apuntador, ahora si recordamos nuestras clases de aritmética de la secundaria, para hacer la suma de dos números complejos hay una regla especial, ya que no se podrá hacer la suma típica que conocemos. Esta suma está dividida en dos partes las cuales se tendrán que sumar los números reales solo con números reales y la suma de números imaginarios.

Ahora, haremos que la variable “**suma**” se le asigne memoria de manera dinámica para trabajar con la cantidad de bytes que solo vamos a requerir, mediante la siguiente línea de código.

/\* suma = (num\_Complejo \*)malloc(sizeof(num\_Complejo)); \*/

Donde el tipo de dato a guardar será de tipo “**num\_Complejo**”.

Luego, lo que haremos es verificar si se le asignó memoria de manera correcta a nuestro apuntador con la siguiente línea de código.

/\*

if (suma == NULL){

printf("Error al asignar memoria para el [num\_Complejo]\n");

exit (0);

}

\*/

Donde sí hubo un error al asignar memoria el programa imprimirá en pantalla el mensaje y se detendrá el programa

Entonces, hacemos la operación para “**suma->real**” en la cual tendrá como operación aritmética la suma de solo los números reales de “**w, z**” apuntando de manera dinámica a la estructura “**num\_Complejo**” y de igual manera para “**suma->imga**” en los valores de “**w, z**” que es la parte imaginaria.

Ejemplo:

suma->real = w->real + z->real;

suma->imga = w->imga + z->imga;

Por último, retornamos el valor de “**suma**”.

NOTA: Declarar la función **num\_Complejo**\* **sumaComplejo (num\_Complejo \*w, num\_Complejo \*z)** como prototipo.

Ahora, utilizaremos nuestra variable “**r**” haciendo referencia a “**resultado**” de las operaciones que vayamos a realizar e irlas guardando en la variable, entonces, “**r**” será a igual a la función “**sumaComplejo (z1, z2)**” teniendo como argumentos las direcciones de memoria de dichos números, ahora imprimiremos en pantalla el valor de dirección de memoria que tiene ahora “**r**” con la ayuda de la función “**muestra\_numComplejo (r)**”.

Después de mostrar el valor de “r” tenemos que liberar su memoria con “free (r)” para obtener el valor del siguiente resultado.

Haremos el mismo procedimiento para la operación “**resta**”.

Esta función tendrá como valor un “**num\_Complejo\* restaComplejo (num\_Complejo \*w, num\_Complejo \*z)**” dentro como argumento de nuestra función recibirá dos parámetros a manera de apuntadores para poder almacenar los datos que realicemos con ellos de manera dinámica. Dentro de nuestra función declaremos una variable de tipo “**num\_Complejo**” con el nombre “**resta**” a manera de apuntador.

Ahora, haremos que la variable “**resta**” se le asigne memoria de manera dinámica para trabajar con la cantidad de bytes que solo vamos a requerir, mediante la siguiente línea de código.

/\* resta = (num\_Complejo \*)malloc(sizeof(num\_Complejo)); \*/

Donde el tipo de dato a guardar será de tipo “**num\_Complejo**”.

Luego, lo que haremos es verificar si se le asignó memoria de manera correcta a nuestro apuntador con la siguiente línea de código.

/\*

if (resta == NULL){

printf("Error al asignar memoria para el [num\_Complejo]\n");

exit (0);

}

\*/

Donde sí hubo un error al asignar memoria el programa imprimirá en pantalla el mensaje y se detendrá el programa

Entonces, hacemos la operación para “**resta->real**” en la cual tendrá como operación aritmética la resta de solo los números reales de “**w, z**” apuntando de manera dinámica a la estructura “**num\_Complejo**” y de igual manera para “**resta->imga**” en los valores de “**w, z**” que es la parte imaginaria.

Ejemplo:

resta->real = w->real - z->real;

resta->imga = w->imga - z->imga;

Por último, retornamos el valor de “**resta**”.

NOTA: Declarar la función **num\_Complejo\* restaComplejo (num\_Complejo \*w, num\_Complejo \*z)** como prototipo.

Ahora, utilizaremos nuestra variable “**r**” haciendo referencia a “**resultado**” de las operaciones que vayamos a realizar e irlas guardando en la variable, entonces, “**r**” será a igual a la función “**restaComplejo (z1, z2)**” teniendo como argumentos las direcciones de memoria de dichos números, ahora imprimiremos en pantalla el valor de dirección de memoria que tiene ahora “**r**” con la ayuda de la función “**muestra\_numComplejo (r)**”.

Después de mostrar el valor de “r” tenemos que liberar su memoria con “free (r)” para obtener el valor del siguiente resultado.

Haremos el mismo procedimiento para la operación “**producto**”.

Esta función tendrá como valor un “**num\_Complejo**\* **producComplejo (num\_Complejo \*w, num\_Complejo \*z)**” dentro como argumento de nuestra función recibirá dos parámetros a manera de apuntadores para poder almacenar los datos que realicemos con ellos de manera dinámica. Dentro de nuestra función declaremos una variable de tipo “**num\_Complejo**” con el nombre “**produc**” a manera de apuntador.

Ahora, haremos que la variable “**produc**” se le asigne memoria de manera dinámica para trabajar con la cantidad de bytes que solo vamos a requerir, mediante la siguiente línea de código.

/\* produc = (num\_Complejo \*)malloc(sizeof(num\_Complejo)); \*/

Donde el tipo de dato a guardar será de tipo “**num\_Complejo**”.

Luego, lo que haremos es verificar si se le asignó memoria de manera correcta a nuestro apuntador con la siguiente línea de código.

/\*

if (produc == NULL){

printf("Error al asignar memoria para el [num\_Complejo]\n");

exit (0);

}

\*/

Donde si hubo un error al asignar memoria el programa imprimirá en pantalla el mensaje y se detendrá el programa

Entonces, hacemos la operación para “**produc->real**” en la cual tendrá como operación aritmética el producto de solo los números reales de “**w, z**” menos el producto de los números imaginarios de “**w, z**” apuntando de manera dinámica a la estructura “**num\_Complejo**” y de igual manera para “**produc->imga**” se hará la operación aritmética del producto real de “**w**” por el imaginario de “**z**” menos el producto del valor de “**w**” imaginaria por el valor real de “**z**”.

Ejemplo:

produc->real = (w->real \* z->real) - (w->imga \* z->imga);

produc->imga = (w->real \* z->imga) + (w->imga \* z->real);

Por último, retornamos el valor de “produc”.

NOTA: Declarar la función **num\_Complejo\* producComplejo (num\_Complejo \*w, num\_Complejo \*z)** como prototipo.

Ahora, utilizaremos nuestra variable “**r**” haciendo referencia a “**resultado**” de las operaciones que vayamos a realizar e irlas guardando en la variable, entonces, “**r**” será a igual a la función “**producComplejo (z1, z2)**” teniendo como argumentos las direcciones de memoria de dichos números, ahora imprimiremos en pantalla el valor de dirección de memoria que tiene ahora “**r**” con la ayuda de la función “**muestra\_numComplejo (r)**”.

Después de mostrar el valor de “r” tenemos que liberar su memoria con “free (r)” para obtener el valor del siguiente resultado.

Haremos el mismo procedimiento para la operación “**división**”.

Esta función tendrá como valor un “**num\_Complejo**\* **diviComplejo (num\_Complejo \*w, num\_Complejo \*z)**” dentro como argumento de nuestra función recibirá dos parámetros a manera de apuntadores para poder almacenar los datos que realicemos con ellos de manera dinámica. Dentro de nuestra función declaremos una variable de tipo “**num\_Complejo**” con el nombre “**divi**” a manera de apuntador.

Ahora, haremos que la variable “**divi**” se le asigne memoria de manera dinámica para trabajar con la cantidad de bytes que solo vamos a requerir, mediante la siguiente línea de código.

/\* divi = (num\_Complejo \*)malloc(sizeof(num\_Complejo)); \*/

Donde el tipo de dato a guardar será de tipo “**num\_Complejo**”.

Luego, lo que haremos es verificar si se le asignó memoria de manera correcta a nuestro apuntador con la siguiente línea de código.

/\*

if (divi == NULL){

printf("Error al asignar memoria para el [num\_Complejo]\n");

exit (0);

}

\*/

Donde si hubo un error al asignar memoria el programa imprimirá en pantalla el mensaje y se detendrá el programa.

Entonces, hacemos la operación para “**divi->real**” en la cual tendrá como operación aritmética el producto de solo los números reales de “**w, z**” más el producto de los números imaginarios “**w, z**” entre el número real “**z**” al cuadrado, más el número imaginario “**z**” al cuadrado, apuntando de manera dinámica a la estructura “**num\_Complejo**” y de igual manera para “**divi->imga**” se hará la operación aritmética del producto imaginario de “**w**” por el real de “**z**” menos el producto del valor real de “**w**” imaginaria por el valor imaginario de “**z**” entre el número real de “**z**” al cuadrado, más el número imaginario de “**z**” al cuadrado.

Ejemplo:

divi->real= ((w->real \* z->real) + (w->imga \* z->imga)) / (pow(z->real, 2) + pow(z->imga, 2));

divi->imga= ((w->imga \* z->real) - (w->real \* z->imga)) / (pow(z->real, 2) + pow(z->imga, 2));

Por último, retornamos el valor de “**divi**”.

NOTA: Declarar la función **num\_Complejo**\* **diviComplejo (num\_Complejo \*w, num\_Complejo \*z)** como prototipo.

Ahora, utilizaremos nuestra variable “r” haciendo referencia a “**resultado**” de las operaciones que vayamos a realizar e irlas guardando en la variable, entonces, “**r**” será a igual a la función “**diviComplejo (z1, z2)**” teniendo como argumentos las direcciones de memoria de dichos números, ahora imprimiremos en pantalla el valor de dirección de memoria que tiene ahora “**r**” con la ayuda de la función “**muestra\_numComplejo (r)**”.

Después de mostrar el valor de “r” tenemos que liberar su memoria con “free (r)” porque nuestras operaciones aritméticas se han completado y ya no necesitaremos la memoria.

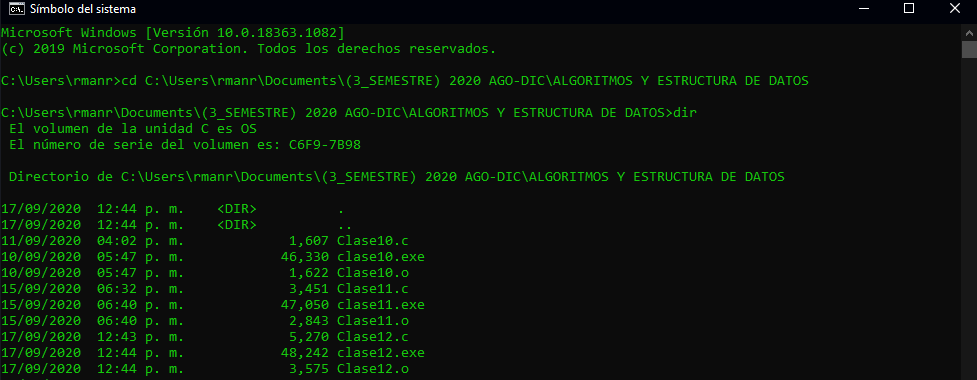
Por último, para limpiar las direcciones de memoria que utilizamos en las variables “**z1, z2, r**” solo escribiremos la palabra “free ()” con el argumento correspondiente a de la variable.

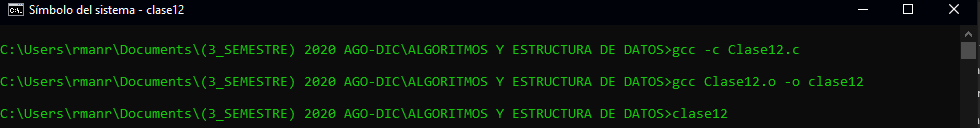
Con esto finalizamos nuestro bloque de código, y ahora nos toca probarlo para observar su funcionamiento.

Para comprobar que nuestro programa realiza lo pedido, abriremos nuestro “**CMD**” he ingresaremos hasta la carpeta donde se tiene guardado el archivo; en mi caso se encuentra en la dirección:

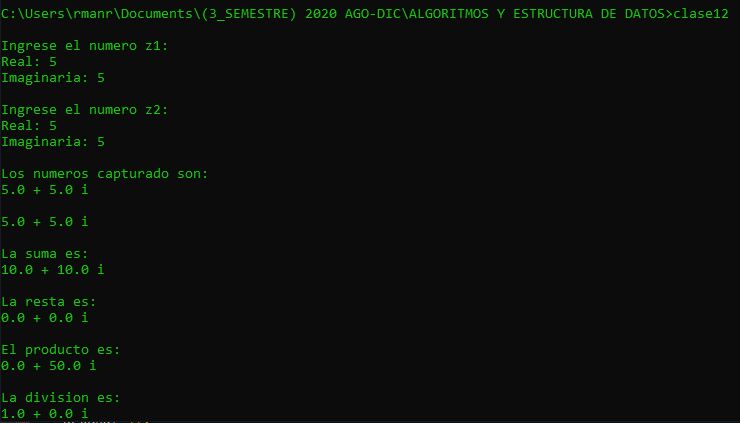
* *C:\Users\rmanr\Documents\(3\_SEMESTRE) 2020 AGO-DIC\ALGORITMOS Y ESTRUCTURA DE DATOS*

Y con el comando <**dir**> veremos que el archivo se ha guardado de manera satisfactoria.



Ahora tendremos que compilar nuestro código con los siguientes comandos que se muestran en la imagen:

Si todo salió bien, solamente dará líneas de salto significando que todo el proceso de compilación y enlazamiento salió bien; de lo contrario, escribiste algún comando mal o tu código tiene algún error de sintaxis.

Ahora ejecutaremos nuestro programa para realizar las pruebas y observar si es lo que se quiso resolver desde un principio:

El programa nos pide ingresar para la variable “**z1**” los valores **reales** e **imaginarios**.

Ingresaremos el valor “5” para **real** e **imaginario**.

De igual manera ingresaremos en la variable “**z2**” los valores **reales** e **imaginarios**.

Ingresaremos el valor “5” para **real** e **imaginario**.

Después se imprimirá en pantalla los valores que hemos ingresado en las dos variables, en las secciones reales e imaginarias.

Luego, se imprimen los valores reales e imaginarias de las operaciones aritméticas que hemos desarrollado en código, para tener una vista mejor de lo que está pasando dentro de cada operación aritmética se explicará en seguida de manera breve.

Suma:

suma->real = w->real + z->real; -------- suma->real = 5 + 5 ------- suma->real = 10

suma->imga = w->imga + z->imga; -------- suma->imga = 5 + 5 ------- suma->imga = 10

La suma es: 10 + 10 i

Producto:

produc->real = (w->real \* z->real) - (w->imga \* z->imga);

produc->real = (5 \* 5) – (5 \* 5)

produc->real = 25 - 25

produc->real = 0

produc->imga = (w->real \* z->imga) + (w->imga \* z->real);

produc->imga = (5 \* 5) + (5 \* 5)

produc->imga = 25 + 25

produc->imga = 50

El producto es: 0 + 50 i

División:

divi->real= ((w->real \* z->real) + (w->imga \* z->imga)) / (pow(z->real, 2) + pow(z->imga, 2));

divi->real = (5 \* 5) + (5 \*5) / (5^2) + (5^2)

divi->real = 50 / 50

divi->real = 1

divi->imga= ((w->imga \* z->real) - (w->real \* z->imga)) / (pow(z->real, 2) + pow(z->imga, 2));

divi->imga = (5 \* 5) – (5 \* 5) / (5^2) + (5^2)

divi->imga = 0 / 50

divi->imga = 0

La división es= 1 + 0 i

Resta:

resta->real = w->real - z->real; -------- resta->real = 5 – 5 -------- resta->real = 0

resta->imga = w->imga - z->imga; -------- resta->imga = 5 - 5 -------- resta->imga = 0

La resta es: 0 + 0 i

Con esto queda resuelto nuestro problema que se planteó desde un principio.

Como conclusión se puede mencionar que es muy fácil de trabajar con apuntadores y memoria dinámica, más en este ejercicio ya que siempre utilizamos el paso de valor mediante referencia, pero en realidad hay una manera mucho más sencilla que es asignar memoria dinámica; para que no se pierda esta práctica es bueno realizar varios ejercicios para no perder la práctica y además aprender más de este tema de memoria.

Este problema me ha sido de mucho agrado ya que hemos utilizado la parte de programación y la parte matemática para llegar a resolver un tema de números complejos, esperemos seguir con este ritmo para aprender más cosas acerca de este tema.