

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:

```

1  #include <stdio.h>
2  #include <stdlib.h> //Librería para usar los apartados de memoria dinamica
3  #include <math.h> //Librería para las funciones matemáticas avanzadas
4
5  //Declaración de la estructura
6
7  typedef struct {
8      float real;
9      float imga;
10
11 } num_Complejo;
12
13 //Esta función regresa una estructura referente a num_Complejo
14 num_Complejo* Captura_numComplejo();
15
16 //Declaración de los prototipos a manera de apuntadores
17 void muestra_numComplejo(num_Complejo *w);
18
19 num_Complejo* sumaComplejo (num_Complejo *w, num_Complejo *z);
20 num_Complejo* restaComplejo (num_Complejo *w, num_Complejo *z);
21 num_Complejo* producComplejo (num_Complejo *w, num_Complejo *z);
22 num_Complejo* diviComplejo (num_Complejo *w, num_Complejo *z);
23
24 int main (void){
25
26     //Las variables serán apuntadores
27     num_Complejo *z1, *z2, *r;
28
29     //Pedimos ingresar los valores reales e imaginarios
30     printf ("\nIngrese el numero z1: ");
31     z1 = Captura_numComplejo();
32
33     printf ("\nIngrese el numero z2: ");
34     z2 = Captura_numComplejo();
35
36     printf ("\n");
37     muestra_numComplejo(z2);
38
39     //Mostrar el resultado de la suma y liberar la memoria
40     r = sumaComplejo (z1,z2);
41     printf ("\n\nLa suma es:");
42     muestra_numComplejo(r);
43     free(r);
44
45     //Mostrar el resultado de la resta y liberar la memoria
46     r = restaComplejo(z1,z2);
47     printf("\n\nLa resta es: ");
48     muestra_numComplejo(r);
49     free(r);
50
51     //Mostrar el resultado de la multiplicacion y liberar la memoria
52     r = producComplejo(z1,z2);
53     printf("\n\nEl producto es: ");
54     muestra_numComplejo(r);
55     free(r);
56
57     //Mostrar el resultado de la división y liberar la memoria
58     r = diviComplejo(z1,z2);
59     printf("\n\nEl producto es: ");
60     muestra_numComplejo(r);
61     free(r);
62
63     //Eliminamos la memoria que ya no volvermos a usar
64     free(z1);
65     free(z2);
66     free(r);
67     getch ();
68     return 0;
69 }

```

```

74 //DivisionComplejo
75 num_Complejo* diviComplejo (num_Complejo *w, num_Complejo *z){
76     num_Complejo *divi; //Declaración de nuestra variable en forma de apuntador
77
78     //Declarar apuntador para guardar la dirección de memoria
79     divi = (num_Complejo *)malloc(sizeof(num_Complejo)); //Cantidad de Bytes que utilizará
80
81     //Verificar si se asignó la memoria correctamente
82     if (divi == NULL){
83         printf("Error al asignar memoria para el [num_Complejo]\n");
84         exit (0);
85     }
86
87     divi->real= ((w->real * z->real) + (w->imga * z->imga)) / (pow(z->real, 2) + pow(z->imga, 2));
88     divi->imga= ((w->imga * z->real) - (w->real * z->imga)) / (pow(z->real, 2) + pow(z->imga, 2));
89
90     return divi;
91 }
92
93 //ProductoComplejo
94 num_Complejo* producComplejo (num_Complejo *w, num_Complejo *z){
95     num_Complejo *produc; //Declaración de nuestra variable en forma de apuntador
96
97     //Declarar apuntador para guardar la dirección de memoria
98     produc = (num_Complejo *)malloc(sizeof(num_Complejo)); //Cantidad de Bytes que utilizará
99
100    //Verificar si se asignó la memoria correctamente
101    if (produc == NULL){
102        printf("Error al asignar memoria para el [num_Complejo]\n");
103        exit (0);
104    }
105
106    produc->real = (w->real * z->real) - (w->imga * z->imga);
107    produc->imga = (w->real * z->imga) + (w->imga * z->real);
108
109    return produc;
110 }
111 //SumaComplejo
112 num_Complejo* sumaComplejo (num_Complejo *w, num_Complejo *z){
113     num_Complejo *suma; //Declaración de nuestra variable en forma de apuntador
114
115     //Declarar apuntador para guardar la dirección de memoria
116     suma = (num_Complejo *)malloc(sizeof(num_Complejo)); //Cantidad de Bytes que utilizará
117
118     //Verificar si se asignó la memoria correctamente
119     if (suma == NULL){
120         printf("Error al asignar memoria para el [num_Complejo]\n");
121         exit (0);
122     }
123
124     suma->real = w->real + z->real;
125     suma->imga = w->imga + z->imga;
126
127     return suma;
128 }
129
130 //RestaComplejo
131 num_Complejo* restaComplejo (num_Complejo *w, num_Complejo *z){
132     num_Complejo *resta; //Declaración de nuestra variable en forma de apuntador
133
134     //Declarar apuntador para guardar la dirección de memoria
135     resta = (num_Complejo *)malloc(sizeof(num_Complejo)); //Cantidad de Bytes que utilizará
136
137     //Verificar si se asignó la memoria correctamente
138     if (resta == NULL){
139         printf("Error al asignar memoria para el [num_Complejo]\n");
140         exit (0);
141     }
142
143     resta->real = w->real - z->real;
144     resta->imga = w->imga - z->imga;
145
146     return resta;
147 }
148
149

```

```

150 void muestra_numComplejo(num_Complejo *w){
151     if (w->imga < 0){
152         printf ("\n%2.1f %2.1f i", w->real, w->imga);
153     }else{
154         printf ("\n%2.1f + %2.1f i", w->real, w->imga);
155     }
156 }
157
158 num_Complejo* Captura_numComplejo(){
159     num_Complejo *w; //Declaración de nuestra variable en forma de apuntador
160
161     //Declarar apuntador para guardar la dirección de memoria
162     w = (num_Complejo *)malloc(sizeof(num_Complejo)); //Cantidad de Bytes que utilizará
163
164     //Verificar si se asignó la memoria correctamente
165     if (w == NULL){
166         printf("Error al asignar memoria para el [num_Complejo]\n");
167         exit (0);
168     }
169
170     //Apuntador hacia estructura w->real
171     printf ("\nReal: ");
172     scanf ("%f", &w->real); fflush (stdin);
173
174     //Apuntador hacia estructura w->imga
175     printf ("Imaginaria: ");
176     scanf ("%f", &w->imga); fflush (stdin);
177
178     return w;
179 }
180

```

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 avanzadas 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 ir las 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 ir las 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 ir las 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:

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

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

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 ir las 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**” e 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.

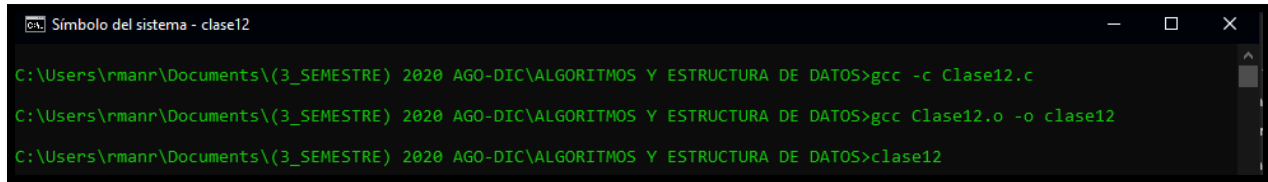
```

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

C:\Users\rmanr\Documents\ (3_SEMESTRE) 2020 AGO-DIC\ALGORITMOS Y ESTRUCTURA DE DATOS>dir
El volumen de la unidad C es OS
El número de serie del volumen es: C6F9-7B98

Directorio de C:\Users\rmanr\Documents\ (3_SEMESTRE) 2020 AGO-DIC\ALGORITMOS Y ESTRUCTURA DE DATOS
17/09/2020  12:44 p. m.    <DIR>          .
17/09/2020  12:44 p. m.    <DIR>          ..
11/09/2020  04:02 p. m.             1,607 Clase10.c
10/09/2020  05:47 p. m.        46,330 clase10.exe
10/09/2020  05:47 p. m.             1,622 Clase10.o
15/09/2020  06:32 p. m.             3,451 Clase11.c
15/09/2020  06:40 p. m.        47,050 clase11.exe
15/09/2020  06:40 p. m.             2,843 Clase11.o
17/09/2020  12:43 p. m.             5,270 Clase12.c
17/09/2020  12:44 p. m.        48,242 clase12.exe
17/09/2020  12:44 p. m.             3,575 Clase12.o
  
```

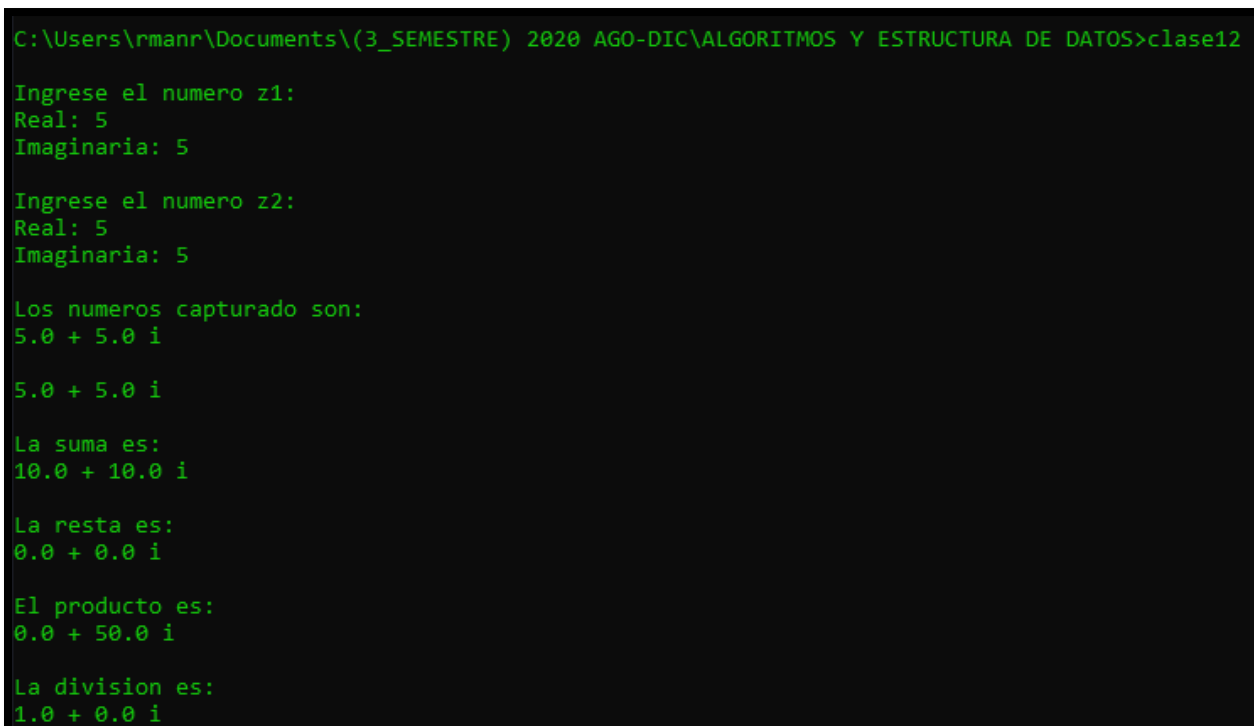
Ahora tendremos que compilar nuestro código con los siguientes comandos que se muestran en la imagen:



```
Símbolo del sistema - clase12
C:\Users\rmanr\Documents\3_SEMESTRE) 2020 AGO-DIC\ALGORITMOS Y ESTRUCTURA DE DATOS>gcc -c Clase12.c
C:\Users\rmanr\Documents\3_SEMESTRE) 2020 AGO-DIC\ALGORITMOS Y ESTRUCTURA DE DATOS>gcc Clase12.o -o clase12
C:\Users\rmanr\Documents\3_SEMESTRE) 2020 AGO-DIC\ALGORITMOS Y ESTRUCTURA DE DATOS>clase12
```

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:



```
C:\Users\rmanr\Documents\3_SEMESTRE) 2020 AGO-DIC\ALGORITMOS Y ESTRUCTURA DE DATOS>clase12
Ingrese el numero z1:
Real: 5
Imaginaria: 5

Ingrese el numero z2:
Real: 5
Imaginaria: 5

Los numeros capturado son:
5.0 + 5.0 i
5.0 + 5.0 i

La suma es:
10.0 + 10.0 i

La resta es:
0.0 + 0.0 i

El producto es:
0.0 + 50.0 i

La division es:
1.0 + 0.0 i
```

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: $10 + 10i$

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: $0 + 50i$

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=  $1 + 0i$ 
```


Resta:

$\text{resta} \rightarrow \text{real} = w \rightarrow \text{real} - z \rightarrow \text{real}; \text{----- } \text{resta} \rightarrow \text{real} = 5 - 5 \text{----- } \text{resta} \rightarrow \text{real} = 0$

$\text{resta} \rightarrow \text{imga} = w \rightarrow \text{imga} - z \rightarrow \text{imga}; \text{----- } \text{resta} \rightarrow \text{imga} = 5 - 5 \text{----- } \text{resta} \rightarrow \text{imga} = 0$

La resta es: $0 + 0i$

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.