



Instituto Politécnico Nacional Escuela Superior de Cómputo

Lenguaje C
Curso de programación en C
A cargo del profesor: Cristhian Avila Sanchez
Escrito por: Adrian González Pardo



Ultima fecha modificado: 18 de diciembre de 2019

Índice

1. Compilación algunos sistemas operativos	3
1.1. Linux	3
1.2. MacOS	3
1.3. Windows	3
1.4. Compilación desde consola	3
2. Pointers (Apuntadores)	3
2.1. Un poco de brevario a la simbología & y *	4
2.2. Aritmetica de apuntadores	4
2.2.1. Algunas cosas antes de jugar mucho con apuntadores	5
3. Apuntadores de varias dimensiones	7
3.1. Ejemplo con apuntadores dobles	7
3.2. Ejemplo con apuntadores de más de dos dimensiones	10
4. Estructura de datos	10
5. Recursividad	13

1. Compilación algunos sistemas operativos

En algunos de los casos más comunes el intentar compilar a nivel consola (CMD, Terminal) es en algunos casos un tabu para los iniciados en la programación o que simplemente se han apoyado de IDE's que les permiten evitarse esta tarea en las cuales si bien en un trabajo rapido, es necesario tambien poder trabajar con este compilador a nivel consola, por lo cual se puede hacer de la siguiente forma en los siguientes sistemas operativos.

1.1. Linux

Si bien el compilador de lenguaje C viene nativamente en los binarios y utilerias del sistema del proyecto Linux y GNU/Linux en algunos casos podemos no encontrar el compilador de lenguaje C (GCC), por lo cual acorde al sistema de empaquetamiento que se maneje en sus sistemas Linux es necesario instalar el paquete gcc, gcc-multilib.

Para el caso de sistemas basados en Debian con empaquetamiento deb y que soporten apt-get, apt se sigue las siguientes líneas de comandos.

```
$_ sudo apt install build-essential gcc gcc-multilib -y
```

1.2. MacOS

En el caso de ser un sistema operativo MacOS, es necesario tomar en cuenta en que las siguientes líneas de comandos puede funcionar o no, por lo cual en caso de que exista algun error es necesario buscar la versión compatible con el sistema operativo que tengan de base

```
$_ sudo port install gcc48
```

```
$_ sudo port select -set gcc mp-gcc48
```

1.3. Windows

Para ultimo caso en un sistema de Windows es necesario bajar el compilador e instalarlo o en su defecto se puede bajar el IDE Codeblocks con el compilador, Dev C o Dev C++, una vez realizada la instalación de alguno de estos IDE's es necesario conocer la ruta de instalación de la aplicación, por lo cual se ira hasta su localización, donde generalmente el compilador en estas aplicaciones esta resguardado en la subcarpeta de ./bin/ , por lo cual copiaremos la ruta completa que este en la navegación, posteriormente en la barra de inicio se buscara "Variables de Entorno", una vez en este apartado en las variables se buscara el apartado de la variable "PATH", en la cual se modificara y a no reemplazando el contenido de esta previamente con ";", en el caso de que sea un string continuo se pegara la ruta (ejemplo) [/ruta/ruta/ideDePreferencia/bin/] y en caso de que no sea un string continuo solo se añadira esto y asi a traves de CMD o PowerShell se puede hacer uso del compilador gcc

1.4. Compilación desde consola

Si bien ya conocemos el como se instala el compilador de lenguaje C para usarlo a nivel consola, ahora un uso generico sin necesidad de quebrarse la cabeza es la compilación rapida de archivos C

```
$_ gcc <archivo de extensión c>
```

Para conocer más opciones acerca de lo que puede o no hacer el binario gcc, tenemos:

```
$_ gcc -help
```

2. Pointers (Apuntadores)

Es una entidad de referencia a otras zonas de memoria, estos hacen una referencia a una variable de un tipo de dato básica (*int*, *float*, *long*, *byte*, *short*)

Un ejemplo de como se escriben estos apuntadores en la lógica del lenguaje C es

```
int *p = NULL,
x=0;
```

De este modo podemos imaginar que se asigna una sección de memoria en estos ambos casos los cuales a nivel memoria contendrán dos direcciones distintas lo cual para términos de ejemplificación se asignara las siguientes secciones de memoria para las variables

```
p = 0x006 , x = 0xAF32
```

Ahora una vez que sabemos que de esta forma se asignan las secciones de memoria, hacemos uso de la asignación de dirección de memoria de la variable x a la variable p, es decir que podamos hacer vía apuntador de la memoria sin necesidad de llegar a la sección de la variable x

```
p = &x;
//Lo cual quiere decir que la dirección del apuntador p esta directamente relacionada a la sección donde habita x sin que exista una forma de apuntar de forma inversa
```

2.1. Un poco de brevario a la simbología & y *

- & : Consulta la dirección en memoria donde reside una variable
- * : Accede al contenido de la variable a la que hace referencia un apuntador

Ahora una vez completado esto podemos pensar que si usamos el apuntador de p pueda añadir o de forma más sencilla asignar un valor a la variable x de la siguiente manera

```
p = &x; //Lo que aqui se señala es que el apuntador p esta relacionado a la dirección de memoria de x
```

Para efectos prácticos diremos que los valores que contienen cada una de estas dos variables p = 0xAF32 // La dirección de memoria de x

x = 0 // El valor almacenado de la variable x

Por otro lado para añadir valores numéricos a la variable x sin necesidad de hacer una asignación como lo es $x = ###$; , lo podemos hacer con una operación de indirección (*)

```
*p = 2019; // Sin que realmente se toque la variable x, se indirecciona el flujo de acceso y se le asigna los valores de 2019 a la variable x
```

Algunas notas al usar apuntadores es necesario también conocer la famosa aritmética de apuntadores.

2.2. Aritmética de apuntadores

```
int *q = NULL,
y=0;
q=&y;
*q=352;
(*q)=(*q)+15; //367
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char *argv[]){
5     int *p=NULL,*q=NULL,*arr=NULL;
6     int x=0,y=0,N=100,k;
7     p=&x;
8     *p=2019;
9     printf("El valor de x = %d \n",x);
10    printf("La direccion de x = %X \n",&x);
11    printf("El valor de p= %X\n",p);
12    printf("El contenido a donde apunta p= %d\n",*p);
13    printf("La direccion de p= %X\n",&p);
14
15    //Aumenta en 1 el contenido de a donde apunta p
```

```

16  (*p)++;
17  printf("1 Valores x= %d\tp= %X\t*p=%d\n",x,p,*p);
18  (*p)*=9;
19  (*p)*=6;
20  (*p)+=3;
21  printf("2 Valores x= %d\tp= %X\t*p=%d\n",x,p,*p);
22  /*Experimento donde puedes matar secciones de memoria completamente
23   * *p++;
24   * printf("Valores x= %d\tp= %X\t*p=%d\n",x,p,*p);
25   * *p++;
26   * printf("Valores x= %d\tp= %X\t*p=%d\n",x,p,*p);*/
27  /*Fuera del experimento se usara un apartado de memoria dinamica arr[N]
28   * Segun la descripcion de malloc tenemos void *malloc(size_t size);
29   * size_t puede ser un valor numerico */
30  arr=(int*)malloc(N*sizeof(int));
31  /*Escritura y lectura convencional*/
32  for(k=0;k<N;k++)
33      arr[k]=3*k;
34
35  for(k=0;k<N;k++)
36      printf("arr[%d]=%d\n",k,arr[k]);
37  /*Finalmente se libera la memoria*/
38  free(arr);
39  return 0;
40 }

```

Código fuente 1

2.2.1. Algunas cosas antes de jugar mucho con apuntadores

Si bien sabemos que el hacer uso de apuntadores nos permite el uso y manejo de memoria dinámica y manejar secciones de memoria acorde al tipo de dato en el que esta especificado, podemos pensar que existen restricciones en las que el mismo lenguaje te puede parar y señalar algunos errores que pueden ocasionar daño lógico al sistema o en su defecto daño a nivel de comunicación de hardware, por ello en algunas nuevas versiones del compilador se le añadieron algunas limitaciones ante el uso o abuso de esta idea.

Por ello acompañado de un pequeño código fuente y con la llamada al operador `sizeof()` podemos conocer a nivel de bytes y bits cuanta memoria ocupa cada tipo de dato en el compilador y de acuerdo a su arquitectura.

```

1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main(void){
5      int a;
6      char b;
7      float c;
8      long d;
9      double e;
10     /*Llamada de sizeof devuelve un valor equivalente en bytes
11     * por ello se multiplica por 8 para hacer la conversion de bytes a bits
12     * por otro lado se hacen llamadas a tipos de datos que estan
13     * nativos del lenguaje por ello los de tipo unsigned y combinaciones
14     * long otherDate no se encuentran en esta lista ya que sizeof no detecta
15     * diferencia alguna. */
16     printf("Size of int = %lu bits is equal to %lu bytes\n"
17           "Size of char = %lu bits is equal to %lu byte\n"
18           "Size of float = %lu bits is equal to %lu bytes\n"
19           "Size of long = %lu bits is equal to %lu bytes\n"
20           "Size of double = %lu bits is equal to %lu bytes\n",
21           sizeof(int)*8,sizeof(int),sizeof(char)*8,sizeof(char),
22           sizeof(float)*8,sizeof(float),
23           sizeof(long)*8,sizeof(long),sizeof(double)*8,sizeof(double));
24     return 0;
25 }

```

Código fuente 2

Ahora una vez que conocemos el cuanto pueden llegar a valer el tipo de dato en el compilador es necesario considerar que el apuntador con el que se esta trabajando brinca de una sección a otra como lo hacen los arreglos estaticos en unidades de 0 a N, pero con la restriccion de que a nivel de memoria como se mostro en el *Código fuente 1* incrementan acorde a su equivalente en bytes, es decir, si un valor *int* se encuentra en la dirección de memoria *0x07E3* un incremento en la posición del apuntador daría como dirección la siguiente (Suponiendo que en la arquitectura que se trabaja el int es equivalente a 4 bytes) *0x07E7*.

Pero que pasa si algún programador comienza a hacer de las suyas con esta aritmética de apuntadores, puede hacer demasiados casting en un arreglo de enteros y de caracteres desde otro tipo de apuntadores, lo cual se muestra a continuación.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void){
5     int p[]={1,2,3,4,5},i;
6     char p1[]={ 'a','b','c','d','e'};
7     void *pv;
8     pv=(void*)p;
9     /*Modo de obtener el tamaño del array solo con operacion sizeof y
10      * conociendo el tipo de dato*/
11     printf("Integer array with void pointer\n");
12     for(i=0;i<((sizeof(p)/sizeof(int)));i++){
13         printf("Void pointer position [%d] address %p that contain %d\n",
14             i,pv,*(int*)pv);
15         pv=(void*)((int*)pv+1);
16     }
17     /*Modo de obtener el tamaño del array solo con operacion sizeof y
18      * sin conocer el tipo de dato*/
19     printf("Char array with void pointer\n");
20     pv=(void*)p1;
21     for(i=0;i<((sizeof(p1)/sizeof(*p1)));i++){
22         printf("Void pointer position [%d] address %p that contain %c\n",
23             i,pv,*(char*)pv);
24         pv=(void*)((char*)pv+1);
25     }
26     return 0;
27 }
```

Código fuente 3

```
Integer array with void pointer
Void pointer position [0] address 0x7ffcae731280 that contain 1
Void pointer position [1] address 0x7ffcae731284 that contain 2
Void pointer position [2] address 0x7ffcae731288 that contain 3
Void pointer position [3] address 0x7ffcae73128c that contain 4
Void pointer position [4] address 0x7ffcae731290 that contain 5
Char array with void pointer
Void pointer position [0] address 0x7ffcae7312a3 that contain a
Void pointer position [1] address 0x7ffcae7312a4 that contain b
Void pointer position [2] address 0x7ffcae7312a5 that contain c
Void pointer position [3] address 0x7ffcae7312a6 that contain d
Void pointer position [4] address 0x7ffcae7312a7 that contain e
```

Output del código de ejemplo 3

Ahora que pasa si por error o juego se hace otro tipo de casting del arreglo.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void){
5     int p[]={1,2,3,4,5},i;
6     char p1[]={ 'a','b','c','d','e'};
7     void *pv;
```

```

8  pv=(void*)p;
9  /*Modo de obtener el tamaño del array solo con operacion sizeof y
10 * conociendo el tipo de dato*/
11  printf("Integer array with void pointer\n");
12  for(i=0;i<((sizeof(p)/sizeof(int));i++){
13      printf("Void pointer position [%d] address %p that contain %d\n",
14             i,pv,*(int*)pv);
15      pv=(void*)((char*)pv+1);
16      /*Cambio de tipo de valor int to char*/
17  }
18  /*Modo de obtener el tamaño del array solo con operacion sizeof y
19 * sin conocer el tipo de dato*/
20  printf("Char array with void pointer\n");
21  pv=(void*)p1;
22  for(i=0;i<((sizeof(p1)/sizeof(*p1));i++){
23      printf("Void pointer position [%d] address %p that contain %c\n",
24             i,pv,*(char*)pv);
25      pv=(void*)((int*)pv+1);
26      /*Segundo cambio char to int*/
27  }
28  return 0;
29 }

```

Código fuente 4

```

Integer array with void pointer
Void pointer position [0] address 0x7ffeca2cb520 that contain 1
Void pointer position [1] address 0x7ffeca2cb521 that contain 33554432
Void pointer position [2] address 0x7ffeca2cb522 that contain 131072
Void pointer position [3] address 0x7ffeca2cb523 that contain 512
Void pointer position [4] address 0x7ffeca2cb524 that contain 2
Char array with void pointer
Void pointer position [0] address 0x7ffeca2cb543 that contain a
Void pointer position [1] address 0x7ffeca2cb547 that contain e
Void pointer position [2] address 0x7ffeca2cb54b that contain 3
Void pointer position [3] address 0x7ffeca2cb54f that contain *
Void pointer position [4] address 0x7ffeca2cb553 that contain *

```

Output del código de ejemplo 4

Esto puede significar que se puede romper o en su defecto realizar operaciones que "no esten permitidas en el lenguaje", pero que dentro de su sintaxis y de la misma curiosidad del programador este puede abusar del movimiento de direcciones y del tipo de casting que existe en el lenguaje.

3. Apuntadores de varias dimensiones

Si bien el hacer uso de apuntadores permite el manejo inteligente de sus variables y de memoria dinámica con apuntadores simples, ahora que pasa cuando se hace en varias dimensiones de apuntador, es decir, que se haga varios apuntadores ""

3.1. Ejemplo con apuntadores dobles

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define OK      1
5  #define ERROR  0
6
7  int  crearVector(int **arr, int N);
8  int  desplegarVector(int *arr, int N);
9  int  destruirVector(int *arr);
10 int  destruirVectorSerguramente(int **arr);
11
12 int  main(int argc, char *argv[]){

```

```

13     int **d= NULL;
14     int *p= NULL, *q= NULL;
15     int *vec= NULL;
16     int x=0, N= 100, sw= OK;
17
18     printf("x= %d\n", x);
19
20     p= &x;
21     *p= 2019;
22
23     printf("x= %d\n", x);
24
25     // apuntador doble:
26     d= &p;
27     q= *d;
28     (*q)++;
29
30     printf("x= %d\n", x);
31
32     sw= crearVector(&vec, N);
33
34     if (sw==OK)
35     {
36         desplegarVector(vec, N);
37         //vec= NULL;
38         //destruirVector(vec);
39         printf("vec= %X\n", vec);
40         destruirVectorSerguramente(&vec);
41         printf("vec= %X\n", vec);
42     }
43
44
45     return 0;
46 }
47
48 int crearVector(int **arr, int N){
49     int *loc= NULL;
50     int k=0;
51
52     loc = (int *) malloc(N*sizeof(int));
53
54     if (loc==NULL)
55         return(ERROR);
56
57     for (k=0; k<N; k++)
58         loc[k]= 6;
59
60     *arr= loc;      // mecanismo de paso por referencia
61     // arr= &loc;   // trabajar con el paso por valor
62                     // i.e copia
63
64     return(OK);
65 }
66
67 int desplegarVector(int *arr, int N){
68     int k=0;
69
70     if (arr==NULL || N<=0)
71         return(ERROR);
72
73     for (k=0; k<N; k++)
74         printf("[%d]= %d\n", k, arr[k]);
75
76     return(OK);
77 }
78
79 int destruirVector(int *arr){

```



```

80     if (arr!=NULL){
81         free(arr);
82         return(OK);
83     }
84     return(ERROR);
85 }
86
87 int destruirVectorSerguramente(int **arr){
88     int *loc= NULL;
89
90     loc= *arr;
91
92     if(loc!=NULL){
93         free(loc);
94         *arr= NULL;
95         return(OK);
96     }
97     return(ERROR);
98 }

```

Código fuente 5

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[]){
5      float **mat= NULL;
6      int M= 10, N= 10;
7      int i=0, j=0, k=0;
8
9      // creacion de matriz:
10
11     mat= (float **) malloc(M*sizeof(float *));
12     //vec= (int *) malloc(N*sizeof(int));
13
14     for (i=0; i<M; i++){
15         mat[i]= (float *) malloc(N*sizeof(float));
16
17         // inicializacion:
18
19         for (i=0; i<M; i++){
20             for (j=0; j<N; j++){
21                 mat[i][j]= i*j;
22
23             for (i=0; i<M; i++){
24                 for (j=0; j<N; j++){
25                     printf("%0.2f  ", mat[i][j]);
26                     printf("\n");
27                 }
28
29                 //destruccion de matriz:
30
31                 for (i=0; i<M; i++){
32                     free(mat[i]);
33                 }
34                 free(mat);
35
36                 return 0;
37             }
38
39             // double **crearMatriz(int M, int N);
40             // void destruirMatriz(double **mat, int M);
41             // inicializarMatriz(double **mat, int M, int N);
42             // multiplicarMatriz(double **mC, double **mA, double **mB, int M, int N, int L)
43             ;
44
45             // int crearMatriz(double ***mat, int M, int N);
46             // int destruirMatriz(double ***mat, int M);

```

```

45
46 // C= A*B;

```

Código fuente 6

3.2. Ejemplo con apuntadores de más de dos dimensiones

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char *argv[]) {
5     double ***vol= NULL;
6     int i=0, j=0, k=0;
7     int M= 10, N= 10, L= 10;
8     vol= (double ***) malloc(L*sizeof(double **));
9
10    for(i=0; i<L; i++){
11        vol[i]= (double **) malloc(M*sizeof(double *));
12        for(j=0; j<M; j++){
13            vol[i][j]= (double *) malloc(N*sizeof(double));
14            for(k=0; k<N; k++)
15                vol[i][j][k]= i*j*k;
16        }
17    }
18    for(i=0; i<L; i++){
19        printf("i= %d\n", i);
20        for (j=0; j<M; j++){
21            for (k=0; k<N; k++)
22                printf("%0.2lf ", vol[i][j][k]);
23            printf("\n");
24        }
25        printf("\n");
26    }
27
28    for (i=0; i<L; i++){
29        for (j=0; j<M; j++)
30            free(vol[i][j]);
31        free(vol[i]);
32    }
33    free(vol);
34    return 0;
35 }

```

Código fuente 7

4. Estructura de datos

Una estructura es una colección de datos, en el no solo contiene datos, sino que igual que contiene una serie de operaciones, en algunos casos un ejemplo en el que se puede trabajar esto es con números complejos, que si bien es claro en algunos lenguajes no esta definido este tipo de dato.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 /* Se redefine el nombre del struct como apuntador de nombre complejo
5  * Mientras que por otro lado se define tComplejo para conocer el tamaño en
6  * Memoria del struct*/
7 typedef struct complejo{
8     /*Definicion x + iy*/
9     float real,imag;
10
11    /*Definicion r * e^ i(ang)*/

```

```

12     float magnitud, fase;
13 }*complejo, tComplejo;
14
15 /*Firmas de las funciones en codigo*/
16 complejo crearNComplejoCar(float,float);
17 complejo crearNComplejoPol(float,float);
18 void destruirComplejo(complejo);
19 void sumarComplejo(complejo,complejo,complejo);
20 void mulComplejo(complejo,complejo,complejo);
21 void desplegarNComplejo(complejo);
22 void desplegarPolarComplejo(complejo);
23 complejo *crearArrayComplejo(int);
24 void destruirArrComplejo(complejo*,int);
25 void initComplejoCar(complejo,float,float);
26 void initComplejoPol(complejo,float,float);
27 void calcularRaicesUn(complejo*,complejo,int);
28
29 int main(int argc, char *argv[]) {
30     complejo zA=NULL, zB=NULL, zC=NULL,*arr;
31     zA=crearNComplejoCar(2.0,3.0); /* z = 2 + 3i*/
32     zB=crearNComplejoPol(5.0,M_PI/3); /* z = 5*e^(PI/3)*i
33     zC=(complejo)malloc(sizeof(tComplejo));
34     int N=10;
35     printf("Suma zA*zB\n");
36     sumarComplejo(zC,zA,zB);
37     desplegarNComplejo(zC);
38     desplegarPolarComplejo(zC);
39     printf("zA\n");
40     desplegarNComplejo(zA);
41     desplegarPolarComplejo(zA);
42     printf("zB\n");
43     desplegarNComplejo(zB);
44     desplegarPolarComplejo(zB);
45     printf("Multiplicacion zA*zB\n");
46     mulComplejo(zC,zA,zB);
47     desplegarNComplejo(zC);
48     desplegarPolarComplejo(zC);
49     printf("zA\n");
50     desplegarNComplejo(zA);
51     desplegarPolarComplejo(zA);
52     printf("zB\n");
53     desplegarNComplejo(zB);
54     desplegarPolarComplejo(zB);
55     printf("Raices complejas\n");
56     arr=crearArrayComplejo(N);
57     calcularRaicesUn(arr, crearNComplejoCar(1.0,0.0),N);
58     destruirArrComplejo(arr,N);
59     destruirComplejo(zA);
60     destruirComplejo(zB);
61     destruirComplejo(zC);
62
63     return 0;
64 }
65
66 /*Escritura de cada una de las funciones y operaciones de la estructura del
67     numero complejo*/
68 complejo crearNComplejoCar(float x,float y){
69     complejo z=NULL;
70     z=(complejo)malloc(sizeof(tComplejo));
71     if(z==NULL){
72         printf("Error al asignar memoria en crearNComplejoCar\n");
73         return NULL;
74     }
75     z->real=x;
76     z->imag=y;
77     z->magnitud=sqrtf((x*x)+(y*y));
78     z->fase=atan2f(y,x);

```

```

78     return z;
79 }
80
81 complejo crearNComplejoPol(float r,float ang){
82     complejo z=NULL;
83     z=(complejo)malloc(sizeof(tComplejo));
84     if(z==NULL){
85         printf("Error al asignar memoria en crearNComplejoPol\n");
86         return NULL;
87     }
88     z->magnitud=r;
89     z->fase=ang;
90     z->real= r*cosf(ang);
91     z->imag=r*sinf(ang);
92     return z;
93 }
94
95 void destruirComplejo(complejo c){
96     if(c!=NULL){
97         free(c);
98     }
99 }
100
101 void sumarComplejo(complejo cC,complejo cA,complejo cB){
102     if(cC==NULL || cA==NULL || cB==NULL){
103         printf("No se puede realizar la operacion especificada\n");
104         return;
105     }
106     cC->real = cA->real + cB->real;
107     cC->imag = cA->imag + cB->imag;
108     cC->magnitud = sqrtf(pow(cC->real,2.0)+pow(cC->imag,2.0));
109     cC->fase = atan2f(cC->imag,cC->real);
110 }
111
112 void mulComplejo(complejo cC,complejo cA,complejo cB){
113     if(cC==NULL || cA==NULL || cB==NULL){
114         printf("No se puede realizar la operacion especificada\n");
115         return;
116     }
117     cC->magnitud=cA->magnitud*cB->magnitud;
118     cC->fase=cA->fase+cB->fase;
119     cC->real= cC->magnitud * cosf(cC->fase);
120     cC->imag= cC->magnitud * sinf(cC->fase);
121 }
122
123 void desplegarNComplejo(complejo z){
124     if(z==NULL){
125         return;
126     }
127     printf("%f %s %fi\t",z->real,(z->imag<0)?(""):("+"),z->imag);
128 }
129
130 void desplegarPolarComplejo(complejo z){
131     if(z==NULL){
132         return;
133     }
134     printf("%f e^(i%f)\n",z->magnitud,z->fase);
135 }
136
137 complejo *crearArrayComplejo(int N){
138     complejo *arr=NULL;
139     int k;
140     arr=(complejo*)malloc(sizeof(complejo)*N);
141     if(arr==NULL){
142         printf("Error al asignar memoria en array\n");
143         return NULL;
144     }

```

```

145     for(k=0;k<N;k++){
146         arr[k]=crearNComplejoCar(0.0,0.0);
147     }
148     return arr;
149 }
150
151 void destruirArrComplejo(complejo *arr,int N){
152     int k;
153     if(arr!=NULL){
154         for(k=0;k<N;k++){
155             destruirComplejo(arr[k]);
156         }
157         free(arr);
158     }
159 }
160
161 void calcularRaicesUn(complejo *root,complejo z,int N){
162     int k=0;
163     float fase,rebanada;
164     if(root==NULL||z==NULL){
165         printf("Error en calcularRaicesUn\n");
166         return;
167     }
168     rebanada= (2.0*M_PI+(z->fase))/N;
169     for(k=0;k<N;k++){
170         if(root[k]==NULL){
171             printf("Error al calcularRaicesUn\n");
172             return;
173         }
174         fase=rebanada*k;
175         initComplejoPol(root[k],z->magnitud,fase);
176         desplegarNComplejo(root[k]);
177         desplegarPolarComplejo(root[k]);
178     }
179 }
180
181 void initComplejoCar(complejo z,float x,float y){
182     if(z==NULL){
183         printf("Error en initComplejoCar\n");
184         return;
185     }
186     z->real=x;
187     z->imag=y;
188     z->magnitud=sqrtf((x*x)+(y*y));
189     z->fase=atan2f(y,x);
190 }
191
192 void initComplejoPol(complejo z,float r,float ang){
193     if(z==NULL){
194         printf("Error en initComplejoCar\n");
195         return;
196     }
197     z->magnitud=r;
198     z->fase=ang;
199     z->real= r*cosf(ang);
200     z->imag=r*sinf(ang);
201 }

```

Código fuente 8

5. Recursividad

Es una entidad que se define en términos de sí mismas, en algunos casos de ellos es:

- Árboles

- Celulas
- Fractales
- Sistemas formales matemáticos
- Computación
 1. Automatización
 2. Máquina de Turing (Máquina Universal)
 - Estados: Q
 - Configuraciones: M
 - Transiciones: δ
 - Estado inicial: q_0
 - Estado final: q_f
 3. Lenguaje que puede ser recursivamente enumerable: \mathcal{L}
 4. Contiene palabras del Lenguaje \mathcal{L} que pueden mapearse al conjunto \mathbb{N}

Un ejemplo de código con recursividad es el siguiente:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 void fun();
4 int main(int argc, char *argv[]) {
5     fun();
6     return 0;
7 }
8
9 void fun(){
10     printf("Invocando fun()\n");
11     fun();
12     printf("Concluyo fun()\n");
13 }
```

Código fuente 9

Si este es ejecutado dependiendo el sistema operativo puede acabar o no

```
Invocando fun()
Invocando fun() x
Invocando fun()
Invocando fun()
Invocando fun()
Invocando fun()
Invocando fun()
Invocando fun()
Invocando fun()
Invocando fun()
Invocando fun()
Invocando fun()
Invocando fun()
Invocando fun()
Invocando fun() el cual contiene una serie de operaciones, en algunos casos un ejemplo es
Invocando fun() y este tipo de dato.
Invocando fun()
Invocando fun()
Invocando fun()
fish: './a.out' terminated by signal SIGSEGV (Address boundary error)
```

Output del código de ejemplo 9
Ejecutado en un sistema operativo Linux

Llevando acabo un contador como en la siguiente forma:

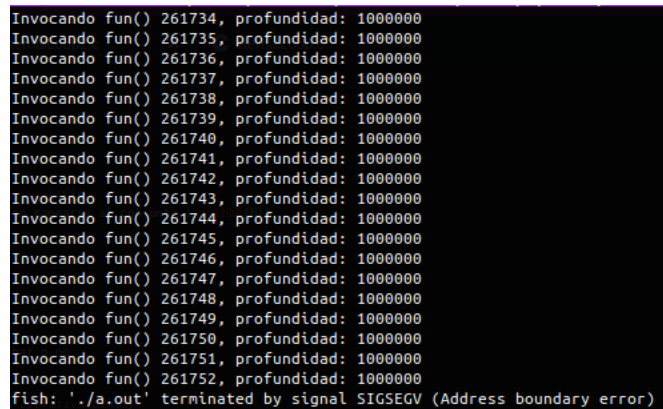
```

1 #include <stdio.h>
2 #include <stdlib.h>
3 void fun(int,int);
4 int main(int argc, char *argv[]) {
5     int profundidad=1000000;
6     if(argc>1){
7         profundidad=atoi(argv[1]);
8     }
9     fun(0,profundidad);
10    return 0;
11 }
12
13 void fun(int nivel,int profundidad){
14     if(nivel==profundidad){
15         return;
16     }
17     printf("Invocando fun() %d, profundidad: %d\n",nivel++,profundidad);
18     fun(nivel,profundidad);
19     printf("Concluyo fun() nivel: %d, profundidad: %d\n",nivel,profundidad);
20 }

```

Código fuente 10

Si este es ejecutado dependiendo el sistema operativo puede acabar o no



```

Invocando fun() 261734, profundidad: 1000000
Invocando fun() 261735, profundidad: 1000000
Invocando fun() 261736, profundidad: 1000000
Invocando fun() 261737, profundidad: 1000000
Invocando fun() 261738, profundidad: 1000000
Invocando fun() 261739, profundidad: 1000000
Invocando fun() 261740, profundidad: 1000000
Invocando fun() 261741, profundidad: 1000000
Invocando fun() 261742, profundidad: 1000000
Invocando fun() 261743, profundidad: 1000000
Invocando fun() 261744, profundidad: 1000000
Invocando fun() 261745, profundidad: 1000000
Invocando fun() 261746, profundidad: 1000000
Invocando fun() 261747, profundidad: 1000000
Invocando fun() 261748, profundidad: 1000000
Invocando fun() 261749, profundidad: 1000000
Invocando fun() 261750, profundidad: 1000000
Invocando fun() 261751, profundidad: 1000000
Invocando fun() 261752, profundidad: 1000000
fish: './a.out' terminated by signal SIGSEGV (Address boundary error)

```

Output del código de ejemplo 10 Ejecutado en un sistema operativo Linux

En algunos casos dependiendo la planificación del sistema operativo, como el hardware del equipo y versión del compilador puede que vaya hasta el final o se pare en otra posición de donde se rompa la recursividad

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 void fun(int,int);
4 void fun1(int,int);
5 void fun2(int,int);
6 int main(int argc, char *argv[]) {
7     int profundidad=1000000;
8     if(argc>1){
9         profundidad=atoi(argv[1]);
10    }
11    printf("Forma 1\n");
12    fun(0,profundidad);
13    printf("Forma 2\n");
14    fun1(0,profundidad);
15    printf("Forma 3\n");
16    fun2(0,profundidad);
17    return 0;

```

```

18 }
19
20 void fun(int nivel,int profundidad){
21     if(nivel==profundidad){
22         return;
23     }
24     printf("Invocando fun() %d, profundidad: %d\n",nivel++,profundidad);
25     fun(nivel,profundidad);
26     printf("Concluyo fun() nivel: %d, profundidad: %d\n",nivel,profundidad);
27 }
28
29 void fun1(int nivel,int profundidad){
30     if(nivel==profundidad){
31         return;
32     }
33     nivel++;
34     fun1(nivel,profundidad);
35     printf("fun1() %d, profundidad: %d\n",nivel,profundidad);
36 }
37
38 void fun2(int nivel,int profundidad){
39     if(nivel==profundidad){
40         return;
41     }
42     printf("fun2() %d, profundidad: %d\n",nivel++,profundidad);
43     fun2(nivel,profundidad);
44 }

```

Código fuente 11

Ejecutando este programa a 5 niveles tenemos

```

Forma 1
Invocando fun() 0, profundidad: 5
Invocando fun() 1, profundidad: 5
Invocando fun() 2, profundidad: 5
Invocando fun() 3, profundidad: 5
Invocando fun() 4, profundidad: 5
Concluyo fun() nivel: 5, profundidad: 5
Concluyo fun() nivel: 4, profundidad: 5
Concluyo fun() nivel: 3, profundidad: 5
Concluyo fun() nivel: 2, profundidad: 5
Concluyo fun() nivel: 1, profundidad: 5
Forma 2
fun1() 5, profundidad: 5
fun1() 4, profundidad: 5
fun1() 3, profundidad: 5
fun1() 2, profundidad: 5
fun1() 1, profundidad: 5
Forma 3
fun2() 0, profundidad: 5
fun2() 1, profundidad: 5
fun2() 2, profundidad: 5
fun2() 3, profundidad: 5
fun2() 4, profundidad: 5

```

Output del código de ejemplo 11