Unidad 1
Funciones, punteros, cadenas, recursividad.

Ing José Luis MARTÍNEZ

22 de mayo de 2020

1. Funciones

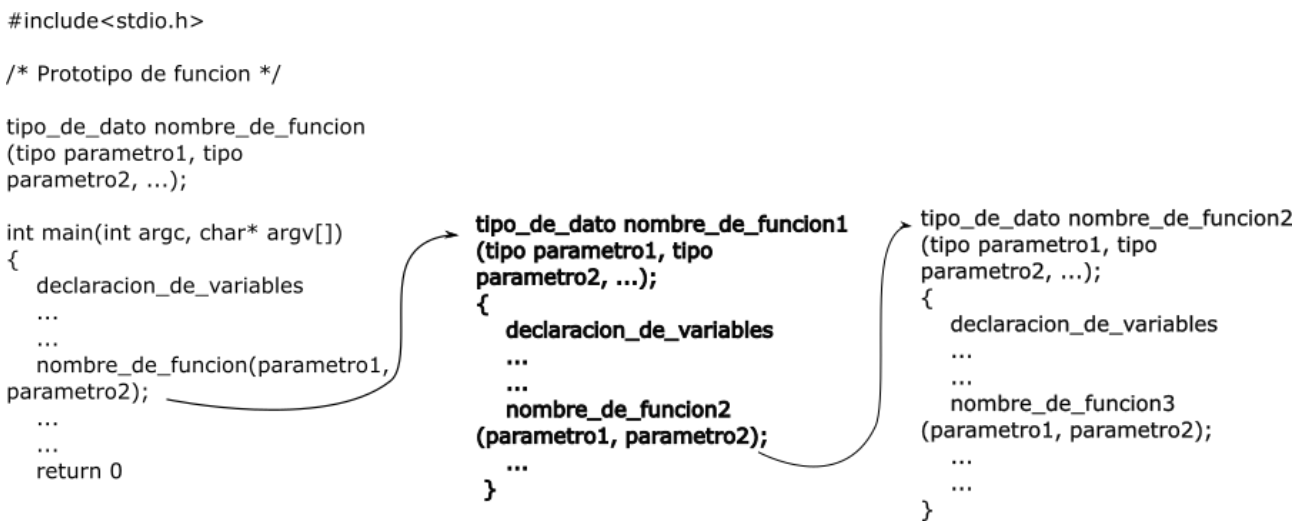
El lenguaje C, como el alumno habrá visto, es un lenguaje basado en funciones que debe tener la función principal *main()* para ser ejecutado. Podemos decir que las funciones son segmentos de programas que ejecutan una o varias tareas. El programador en C debe acostumbrarse a utilizarlas por los siguientes motivos:

1. Brindan claridad en la codificación
2. Permiten reutilizar el software.
3. Permiten aplicar la estrategia de divide y triunfarás, separando un programa complejo en varios problemas más simples.
4. Transportabilidad del código, ya que al solucionar un problema, si se lo hace lo suficientemente general se lo puede llevar a otros programas
5. Posibilidad de crear bibliotecas de funciones, para evitar estar repitiendo programas

El lenguaje C, tiene la función main(), que es la encargada de llamar a las otras funciones que realizarán las acciones, tendrá la forma

```
#include<stdio.h>

/* Prototipo de funcion */

tipo_de_dato nombre_de_funcion(tipo parametro1, tipo parametro2, ...);

int main(int argc, char* argv[])
{
    declaracion_de_variables
    ...
    ...
    nombre_de_funcion(parametro1, parametro2);
    ...
    ...
    return 0

}
```

Una función también puede llamar a otra función



Los valores de los parámetros se le pueden pasar a la función de dos formas

1. **Por valor:** Los valores de las variables son pasados desde la función invocadora (la que llama a la otra función, por ej. main()) a la función invocada.
2. **Por referencia:** Una variable en C tiene asociada un nombre y un valor, al momento de declararla se reserva la memoria necesaria para alojarla, ese lugar de memoria tiene una dirección única para cada variable. En la llamada por referencia las direcciones de las variables son pasadas desde la función invocadora a la función invocada.

Ejemplo. DUP que pase los coeficientes de una ecuación de segundo grado, y devuelva el valor del discriminante. Inicialize el valor del discriminante en la función main(), y verifique si cambiaron luego de invocar a la función, pasando los parámetros por valor y por referencia.

```
#include<stdio.h>

/* Prototipo de funcion */

void discValor(float a, float b, float c, float D);
void discRef(float* a, float* b, float* c, float* D);

int main(int argc, char* argv[])
{
    float a = 1.0, b = 1.0, c = 1.0, D=0.0;
    printf("El valor de D antes de llamar a la funcion por valor es: %f\n", D);
    discValor(a, b, c, D);
    printf("El valor de D despues de llamar a la funcion por valor y \n antes de llamarla por
    discRef(&a, &b, &c, &D);
    printf("\nEl valor de D despues de llamar a la funcion por por referncia : %f.\n", D);

    getchar();
    return 0;

}

void discValor(float a, float b, float c, float D)
{
    D=b*b - 4.0*a*c;
    printf("\nEl valor de D dentro de la funcion por valor es %f.\n", D);
}
```

```
void discRef(float *a, float *b, float *c, float *D)
{
    *D = (*b)*(*b) - 4.0*(*a)*(*c);
    printf("\nEl valor de D dentro de la funcion por referencia es %f.\n", D);//
}
```

2. Breve introducción a la programación modular

Cuando se escribe un código en lenguaje C es deseable que si se tienen funciones que pueden servir para algún otro programa, poder disponer de ellas sin la necesidad de estar buscando en cada programa para copiarla y pegarla.
Esto representa un gran inconveniente al momento de reutilizar el código, además la extensión del código puede llegar a ser muy grande y difícil de leer.
Con C podemos solucionar estos inconvenientes creando bibliotecas de funciones que tengan alguna característica, que deba incluir en los programas que diseñe. Debemos entonces por un lado separar la biblioteca de la función principal de aplicación, guardando las funciones de biblioteca en archivos distintos.
Tendremos en consecuencia que generar dos tipos de archivos:

1. **Encabezado:** Es un archivo que se distingue por tener la extensión .h, en él se declarar los prototipos de función, constantes y valores a utilizar en el encabezado o en las funciones, tiene la siguiente forma

   ```
   #ifndef AREAS_H_INCLUDED
   #define AREAS_H_INCLUDED
   ... definir bibliotecas a utilizar
   ... definir constantes
   ... declarar prototipos de función
   #endif // AREAS_H_INCLUDED
   ```

2. **Biblioteca:** Es un archivo con las funciones que corresponden a los prototipos declarados en el encabezado.

Con la directiva *#include .*^{en}*cabezado.h "*, se agrega la biblioteca a la función principal o también a otra biblioteca que quiera definir

Ejemplo. Construya una biblioteca de funciones para calcular las áreas de las figuras geométricas y luego aplíquelas en un programas.

Encabezado

```
/* Encabezado Areas.h */
#ifndef AREAS_H_INCLUDED
#define AREAS_H_INCLUDED
#include<math.h>
#define PI 3.141592654

/* Prototipos de funciones */
float rectangulo(float a, float b);

float trianguloBH(float a, float b);
```

```
float trianguloABC(float a, float b, float c);

float circulo(float r);

#endif // AREAS_H_INCLUDED
```

Bioteca de funciones.

```
/* Biblioteca de funciones Areas.c, para el
cálculo de areas de distintas figuras geométricas */

#include "Areas.h"
//#include<math.h>
//#define PI 3.141592654

/* Rectangulo */

float rectangulo(float a, float b)
{
    return a*b;
}

/* Triangulo del que sabemos la altura */

float trianguloBH(float a, float b)
{
    return a*b/2;
}

/* triangulo del que conocemos la longitud de sus lados, se aplica la formula
de Herón  A= sqrt(s(s-a)(s-b)(s-c)), s=(a+b+c)/2 */
float trianguloABC(float a, float b, float c)
{
    float s;
    s=(a + b + c)/2;
    return sqrt(s*(s-a)*(s-b)*(s-c));
}

/* Circulo */
float circulo(float r)
{
    return PI * r * r;
}
```

Función de implementación

```
/* usoAreas.c, utiliza la biblioteca Areas */

#include<stdio.h>
#include "Areas.h"

int main(int argc, char *argv[])
{
```

```
    float areaRec, areaCirc;
    areaRec = rectangulo(7, 6);
    printf("El area del rectangulo es: %.2f\n", areaRec);
    areaCirc = circulo(10);
    printf("El area del circulo es: %.2f\n", areaCirc);
    return 0;
}
```

Compilación. Para compilar los archivos multificheros si estamos en linux escribimos en la línea de comandos:

```
gcc -Wall ficheroA.c ficheroB.c <todos los archivos que quiera agregar> -o salida
```

En cambio si nos encontramos dentro de un IDE, debemos colocar todos los archivos dentro de un proyecto.

2.1. Problemas funciones.

1. Modificar el programa de las áreas de las figuras geométricas, para que contemple la mayor cantidad de casos posibles
2. Cree una biblioteca de funciones que permita calcular las operaciones con matrices, es decir sumar, restar, multiplicar, obtener inversa, trapuesta, etc...

3. Punteros

Los punteros son direcciones a variables, al definirla podemos asociarle un puntero que contenga su ubicación, y desde él acceder a la variable desde cualquier lugar del programa. La programación utilizando punteros tiene varias ventajas

1. Pasar información entre funciones en ambos sentidos
2. Permiten devolver múltiples datos desde una función, a través de sus argumentos
3. Proveen una forma alternativa de acceder a los elementos de un array.
4. Se los utiliza para pasar arreglos y cadenas como argumentos de función
5. Se los utiliza para crear estructuras de datos complejas como son las listas enlazadas, pilas, colas, árboles y grafos
6. Son utilizados para la locación dinámica de memoria

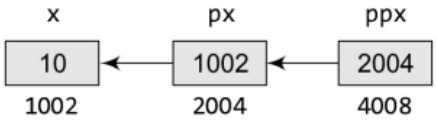
Ejemplo. DUP que dadas los valores de cargas eléctricas discretas y ubicadas en un sistema de coordenadas cartesianas, calcule el campo eléctrico en el origen

```
/* El campo electrico en un punto del espacio debido a una carga electrica
está dado por Ex = k q/|r|^3 (x i);  Ey = k q/|r|^3 (y j)
 r^2 = x^2 + y^2 */

 #include<stdio.h>

 #include<math.h>
 #define k 9e9

 void campo(float *ptrEx, float *ptrEy, int n);
```

```
 void calculaCampo(float *ptrEx, float *ptrEy, float q, float x, float y);
 int main(int argc, char* argv[])
 {
     int n;
     float accEx, accEy, Ex, Ey, *ptrEx, *ptrEy;
     ptrEx = &Ex;
     ptrEy = &Ey;
     printf(" Ingrese la cantidad de cargas que va a utilizar: \n");
     scanf("%d", &n);
     campo(ptrEx, ptrEy, n);

     printf("\nEl valor de la suma de las componentes en x es: %f. \n", Ex);
     printf("\nEl valor de la suma de las componentes en y es: %f. \n", Ey);

     printf("\n El módulo del campo eléctrico es: %f", sqrt(Ex*Ex + Ey * Ey));
   printf("\n El ángulo del campo eléctrico es: %f", atan(Ey/Ex));


     getchar();
     return 0;
 }

void campo(float *ptrEx, float *ptrEy, int n)
{
    float q;
    float x,y, accEx=0.0, accEy=0.0;
    while (n > 0)
    {
        printf("Ingrese el valor de la carga q(%d): ", n);
        scanf("%f", &q);
        printf("Ingrese la coordenada x: ");
        scanf("%f", &x);
        printf("Ingrese la coordenada y: ");
        scanf("%f", &y);
        calculaCampo(ptrEx, ptrEy, q, x, y);
        accEx += *ptrEx;
        accEy += *ptrEy;
        printf("\nEl valor de la suma de las componentes en x es: %f. \n", accEx);
        printf("\nEl valor de la suma de las componentes en y es: %f. \n", accEy);
        --n;
    }
    *ptrEx = accEx;
    *ptrEy = accEy;

}

void calculaCampo(float *ptrEx, float *ptrEy, float q, float x, float y)
{
    float r2= x*x + y*y;
    float modr=sqrt(r2);
    float modr3 = pow(modr,3);
    *ptrEx = (k*q/modr3)*x;
    *ptrEy = (k*q/modr3)*y;
```

```
}
```

3.1. Punteros a punteros.

un puntero puede apuntar a otro puntero de la forma 


```
  int x = 10;
int *px, **ppx;
px = &x;
ppx = &px;
...
...
printf("\n %d", **ppx);
```

3.2. Punteros a arreglos.

Como se ha estudiado un arreglo es una colección de valores de un tipo de datos, donde el nombre del arreglo es la dirección del primer dato y los datos sucesivos se escriben en direcciones sucesivas. En consecuencia se puede acceder a un arreglo mediante punteros. En este caso no necesito definir una variable de tipo puntero porque el arreglo es un puntero en sí mismo.

Ejemplo. Pase un arreglo de caracteres a una función y que desde esta se modifique un elemento

```
int main(int argc, char *argv[])
{
    char c[]= {'m','e','s','a'};

    printf("El valor de c antes de llamar a la funcion cambiaLetra().\n");
    for(int i = 0; i < 4; i++)
        printf("%c", c[i]);
    printf("\n");

    cambiaLetra(c);

    printf("El valor de c despues de llamar a la funcion cambiaLetra().\n");
    for(int i = 0; i < 4; i++)
        printf("%c", c[i]);
    printf("\n");

    return 0;
}

void cambiaLetra(char *c)
{
    c[1] = 'a';
}
```

3.3. Punteros en los arrays de dos dimensiones

Para apuntar a un array bidimensional como tal,o lo que es lo mismo, para apuntar a su inicio, el compilador de C++ considera que un array bidimensional es en realidad un array de punteros a los arrays que forman sus filas. Por tanto, será necesario un puntero doble o puntero a puntero, que contendrá la dirección del primer puntero del array de punteros a cada una de las filas del array bidimensional o matriz. Si a se ha definido como un array bidimensional, el nombre del array a es un puntero constante que apunta a la primera fila a[0] . El puntero a+1 apunta a la segunda fila a[1] , etc. A su vez a[0] es un puntero que apunta al primer elemento de la fila 0 que es a[0][0] . El puntero a[1] es un puntero que apunta al primer elemento de la fila 1 que es a[1][0] , etc.

Representamos un array de dos dimensiones

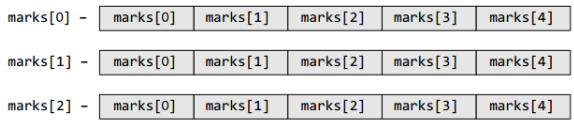


Figura 1: Array 2D

Representación en memoria del Array 2D

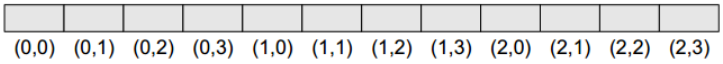


Figura 2: Array 2D

Ejemplo Array bidimensional, punteros y posiciones de memoria. Dada la declaración float A[5][3] que define un array bidimensional de cinco filas y tres columnas, se tiene la siguiente estructura:

Puntero a puntero fila		**Puntero a fila**		**Array bidimensional** float A[4][3]		
A	→	A[0]	→	A[0][0]	A[0][1]	A[0][2]
A+1	→	A[1]	→	A[1][0]	A[1][1]	A[1][2]
A+2	→	A[2]	→	A[2][0]	A[2][1]	A[2][2]
A+3	→	A[3]	→	A[3][0]	A[3][1]	A[3][2]
A+4	→	A[4]	→	A[4][0]	A[4][1]	A[4][2]

A es un puntero que apunta a un array de 5 punteros A[0], A[1], A[2], A[3], A[4] .
A[0] es un puntero que apunta a un array de tres elementos A[0][0], A[0][1], A[0][2] .
A[1] es un puntero que apunta a un array de tres elementos A[1][0], A[1][1], A[1][2] .
A[2] es un puntero que apunta a un array de tres elementos A[2][0], A[2][1], A[2][2] .
A[3] es un puntero que apunta a un array de tres elementos A[3][0], A[3][1], A[3][2] .
A[4] es un puntero que apunta a un array de tres elementos A[4][0], A[4][1], A[4][2] .
A[i][j] es equivalente a las siguientes expresiones:

- $*(A[i]+j)$ el contenido del puntero a la fila i más el número de columna j.
- $*((*(A+i))+j)$. Si se cambia $A[i]$ por $*(A+i)$ se tiene la siguiente expresión anterior.
- $*(\&A[0][0]+3*i+j)$.

Ejemplo Escriba un programa que acceda a los miembros de un arreglo mediante aritmética de punteros.

```
#include <stdio.h>

int main(int argc, char* argv[])
```

```
{
    int matrizA[3][3];
    for(int i=0; i<3; i++)
    {
        for(int j=0; j<3; j++)
        {
            printf("Ingrese la fila %d y la columna %d. \n", i, j);
            scanf("%d", &matrizA[i][j]);
        }
    }

    for (int i=0; i<3; i++)
    {
        printf("\nEsta en la fila i = %d \n", i);
        for(int j=0; j<3 ; j++)
        {
            printf(" \t columna j = %d, matrizA[%d][%d] = %d", j, i, j, matrizA[i][j]);
        }
    }

    printf("\n\t Prueba con *(matrizA[i]+j)\n");
    for (int i=0; i<3; i++)
    {
        for(int j=0; j<3; j++)
        {
            printf("%d \t", *(matrizA[i]+j) );
        }
        printf("\n");
    }

    printf("\n\t Prueba con *(*(matrizA + i) + j)\n");
    for(int i=0; i<3;i++)
    {
        for(int j=0; j<3; j++)
        {
            printf("%d \t", *(*(matrizA +i)+j));

        }
        printf("\n");
    }

    printf("\n\t Prueba con *(&A[0][0]+ 3*i+j)\n");
    for(int i=0; i<3 ; i++)
    {
        for(int j=0; j<3;j++)
        {
            printf("%d \t",*(&matrizA[0][0]+ 3*i+j));
        }
        printf("\n");
    }

    return 0;
}
```

A es un puntero que apunta a A[0].
A[0] es un puntero que apunta a A[0][0].
Si A[0][0] se encuentra en la dirección de memoria 100 y teniendo en cuenta que un *float* ocupa 4 bytes, la siguiente tabla muestra un esquema de la memoria:

Contenido de puntero a puntero fila	**Contenido de puntero a fila**	**Direcciones del array bidimensional float A[4][4]**		
*A = A[0]	A[0] = 100	A[0][0] = 100	A[0][1] = 104	A[0][2] = 108
*(A+1) = A[1]	A[1] = 112	A[1][0] = 112	A[1][1] = 116	A[1][2] = 120
*(A+2) = A[2]	A[2] = 124	A[2][0] = 124	A[2][1] = 128	A[2][2] = 132
*(A+3) = A[3]	A[3] = 136	A[3][0] = 136	A[3][1] = 140	A[1][2] = 144
*(A+4) = A[4]	A[4] = 148	A[4][0] = 148	A[4][1] = 152	A[4][2] = 156

3.4. Paso de arreglos a funciones por referencia

Considere un arreglo de dos dimensiones

```
int mat[5][5];
```

Para declarar un puntero de un array bidimensional, se escribe

```
int **ptr
```

Aquí **ptr es un arreglo de punteros (a un arreglo de una dimensión) mientras que

```
int mat[5][5];
```

es un arreglo bidimensional. Por lo tanto no son lo mismo y no son intercambiables. Los elementos del arreglo se pueden acceder mediante:

```
mat[i][j]
*(*(mat + i) + j)
*(mat[i]+j);
```

Para especificar en el prototipo de función que vamos a utilizar punteros podemos hacer

```
void funcionArray(int ptr[][columnas]);
```

o también

```
void funcionArray(int (*)[10]);
void funcionArray(int (*ptr)[10]);
```

Observe que en ptr[][columnas] el primer corchete vacío está indicando que se trata de un puntero, las sintaxis (*)[10] y (*ptr)[10] indican que son arreglos bidimensionales de 10 columnas, no confundir con *ptr[10] que es un arreglo unidimensional de 10 elementos.

El siguiente código ilustra el uso de punteros a un arreglo bidimensional.

```
#include <stdio.h>
int main()
{
int arr[2][2]={{1,2}, {3,4}};
int i, (*parr)[2];
parr = arr;
```

```
for(i = 0; i < 2; i++)
{
 for(j = 0; j < 2 ;j++)
 printf(" %d", (*(parr+i))[j]);
}
return 0;
}
```

Ejemplo. Escriba un programa en C que modifique por referencia un arreglo de 2D.

```
/* Pasar un array 2D por referencia*/

#include<stdio.h>

void cargaArray2D(float (*D2)[3]);

int main(int argc, char *argv[])
{
    float array2D[3][3]={0};

    printf("El array2D antes de entrar a la funcion. \n");

    for(int i=0; i<3; i++)
    {
        for(int j=0; j < 3; j++)
        {
            printf("%f  ", array2D[i][j]);
        }
        printf("\n");
    }

    printf("\n");

    cargaArray2D(array2D);

    printf("El array2D despues de entrar a la funcion. \n");

    for(int i=0; i<3; i++)
    {
        for(int j=0; j < 3; j++)
        {
            printf("%f  ", array2D[i][j]);
        }
        printf("\n");
    }

    printf("\n");


    return 0;
}

void cargaArray2D(float (*D2)[3])
```

```
{
    for(int i=0; i<3; i++)
    {
        for(int j=0; j < 3; j++)
        {
            D2[i][j] = (float) i+j;
        }

    }

}
```

3.5. Problemas con punteros.

4. Modifique el problema 1 para utilizarlo con punteros.
5. Modifique el problema 2 para utilizarlo con punteros.

4. Manipulación de cadenas

Una cadena es un tipo de dato compuesto, un array de caracteres *(char)*, terminado por un carácter nulo ('\0'), *NULL*. Un ejemplo es "*ABC*". Cuando la cadena aparece dentro de un programa se verá como si se almacenarán cuatro elementos: 'A' , 'B' , 'C' y '\0' . En consecuencia, se considerará que la cadena "*ABC*" es un array de cuatro elementos de tipo char.El valor real de esta cadena es la dirección de su primer carácter y su tipo es un puntero a char. Aplicando el operador * a un puntero a char se obtiene el carácter que forma su contenido; es posible también utilizar aritmética de direcciones con cadenas:

$*"ABC"$	es igual a	'A'
$*("ABC"+1)$	es igual a	'B'
$*("ABC"+2)$	es igual a	'C'
$*("ABC"+3)$	es igual a	\0

De igual forma, utilizando el subíndice del array se puede escribir:

$"ABC"[0]$	es igual a	'A'
$"ABC"[1]$	es igual a	'B'
$"ABC"[2]$	es igual a	'C'
$"ABC"[3]$	es igual a	\0

La finalización de la cadena con el caracter \0 no se da en el caso de un char.

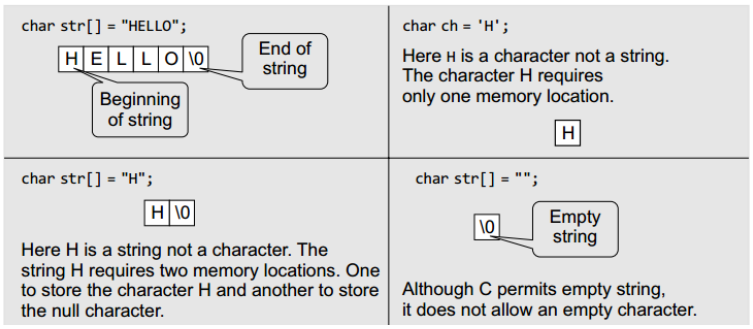


Figura 3: Diferencia entre cadena y caracter

La manipulación de cadenas se realiza mediante el archivo de cabecera <string.h>. Las funciones de la librería y como utilizarlas se pueden encontrar en `http://c.conclase.net/librerias/?ansilib=string#inicio`

```
* El siguiente programa lee y escribe el nombre y legajo de un alumno.*/

#include <stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include <string.h>
int main(int argc, char *argv[])
{
char Apellidos [ 32];
char Nombres [32];
char Legajo[6];
int i=0;
//int numLegajo;

printf("\nNombres: " ) ; fgets (Nombres, 32, stdin);
printf("\nApellidos: " ) ; fgets (Apellidos, 32, stdin);
printf("\nLegajo: "); fgets(Legajo, 6, stdin);

//memcpy(Nombres, Nombres, strlen(Nombres)-1);  // prueba de copiar todo menos \0
strcpy(Nombres, Nombres);
//for(int i=0; i < strlen(Nombres)-2; i++)  // otra prueba fallida
//    Nombres[i]=Nombres[i];

/* visualizar cadenas */
printf ("\n\n %s \t %s \n", Nombres, Apellidos);  // fgets carga el \0 por imprime en renglone
printf ("%s \n", Legajo);

/* Eliminar el caracter de fin de linea \0*/
for(int i=0; i<strlen(Nombres)-1; i++)
    putchar(Nombres[i]);
printf("\t");
for(int i=0; i<strlen(Apellidos)-1; i++)
    putchar(Apellidos[i]);

/* Ver el tamaño de una cadena */
printf("\nLa cadena Nombres tiene un tamagno de %d", strlen(Nombres));
printf("\nLa cadena Nombres contando un campo a la vez es");
while(Nombres[i] != '\n')
        i++;
printf(" %d: \n", i);


/* Convertir un string en un numero con atoi(), para convertir a flotante se usa atof() */
printf("\nEl numero de legajo es %d. ", atoi(Legajo));  //función de la libreria stdlib.h
printf("\nEl numero de legajo siguiente es %d. \n", atoi(Legajo) + 1);

/* Convertir los caracteres de la librería en mayusucula*/
i=0;
while(Apellidos[i] != '\0')
{
    Apellidos[i] = toupper(Apellidos[i]);
    i++;
}
```

```
printf("\nEl apellido en mayusculas es : %s", Apellidos);

/* Concatenar cadenas */

printf("\nEl nombre completo es: %s", strcat(Nombres, Apellidos));

/* Comparar cadenas y copiar valores*/

if(memcmp(Nombres, "Maria Rosa", 10) != 0)
    //strcpy(Nombres, "Maria Rosa ");  // peligrosa porque no se verifica la longitud al igual
    strncpy(Nombres, "Maria Rosa ", 11);
printf("El resultado de la comparacion es %d ", memcmp(Nombres, "Maria Rosa", 10));
printf("\nEl nombre completo es: %s", strcat(Nombres, Apellidos));

return 0;

}
```

4.1. Problemas con cadenas

1. Escribe un programa que acepte una cadena de caracteres (que podrá contener cualquier carácter a excepción del retorno de carro) y que diga cuántas vocales contiene.

2. Escribe un programa que acepte una cadena de caracteres (que podrá contener cualquier carácter a excepción del retorno de carro) y que le escriba al revés.

3. Escribe un programa que pida dos cadenas de caracteres al usuario y que informe si ambas cadenas son iguales (se considera que las cadenas son iguales aunque difieren en el uso de letras mayúsculas y minúsculas), que imprima por pantalla el resultado de concatenar ambas cadenas de caracteres y los 15 primeros caracteres de la cadena concatenada.

4. Escribe un programa que devuelva el número de caracteres que hay entre la primera y la última aparición de un carácter dado en una cadena.

5. Escribe un programa que lee una cadena de caracteres de teclado e indique si es no palíndroma (se lee igual de izquierda a derecha que de derecha a izquierda, sin tener en cuenta los espacios en blanco y las mayúsculas). Por ejemplo: "dábale arroz a la zorra el abad".

5. Recursion

Como se ha estudiado el lenguaje C basa su funcionamiento en las funciones, estas funciones pueden llamar a otras funciones, cada vez que lo hace los valores de sus variables son almacenados para, al finalizar la tarea la función invocada, retomarlas para continuar con el programa.
Entre las funciones invocadas una muy especial es ella misma, es decir que la función se puede llamar a sí misma, a este proceso se le denomina **recursión**.

Una función recursiva es una función que continuamente se llama a sí misma hasta encontrar una condición de finalización, también llamado caso **base**, a partir del cual comienza a desandar el camino devolviendo los valores guardados en la pila. OJO el caso base debe estar siempre presente para evitar una recursión sin fin.

Por ejemplo el siguiente programa implementa una función recursiva que imprime un número y su ubicación en memoria en cada caso, al llegar al caso base desanda el camino e imprime el número y la ubicación en memoria en forma inversa.

```
/* Ejemplo de recursion imprime número y ubicación de memoria ascendente y descendente*/
#include <stdio.h>

void up_and_down(int);

int main(void)
{
 up_and_down(1);
 return 0;
}
void up_and_down(int n)
{
 printf("Nivel %d: n ubicacion en memoria %p\n", n, &n); /* 1 */
 if (n < 4)
 up_and_down(n+1);
 printf("Nivel %d: n ubicacion en memoria %p\n", n, &n); /* 2 */
}
```

La función main() también puede ser recursiva, como se muestra en el siguiente ejemplo

```
#include<stdio.h>


int main( )
{
 int c ;

 if ( (scanf("%d", &c)) != EOF )
 {
     main( );
     printf("%d ", c);
 }
 return 0;
}
```

5.1. Tipos de recursión

1. Recursión lineal.
   - *a*) Recursión lineal no final.
   - *b*) Recursión lineal final.
2. Recursión múltiple.
3. Recursión mutua.

5.2. Recursión lineal

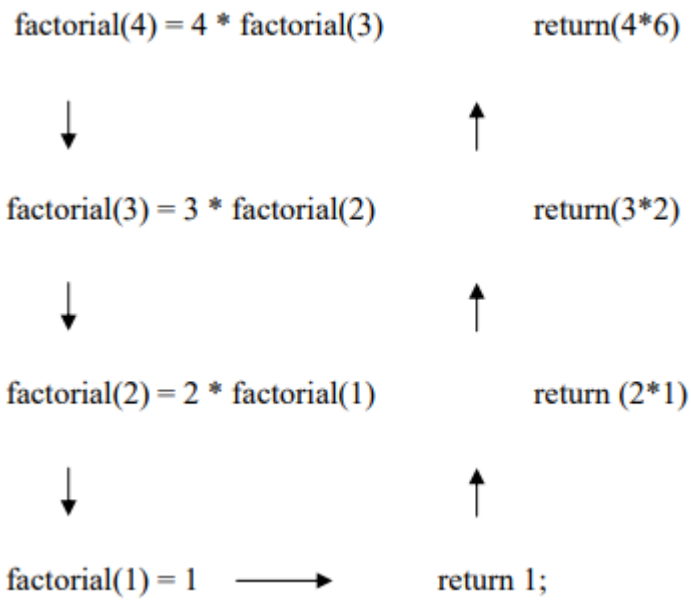
En la recursión lineal cada llamada recursiva genera, como mucho, otra llamada recursiva. Se pueden distinguir dos tipos de recursión lineal atendiendo a cómo se genera resultado.

5.2.1. Recursión lineal no final

En la recursión lineal no final el resultado de la llamada recursiva se combina en una expresión para dar lugar al resultado de la función que llama. El ejemplo típico de recursión lineal no final es cálculo del factorial de un número ($n! = n*(n-1)*\cdots*2*1$). Dado que el factorial de un número n es igual al producto de n por el factorial de $n-1$, lo más natural es efectuar una implementación recursiva de la función factorial. Veamos una implementación de este cálculo:

```
/* Cálculo del factorial de un número */

#include <stdio.h>

int factorial(int numero)
{
  if (numero > 1) return (numero*factorial(numero-1));
  else return(1);
}
int main (int argc, char* argv[])
{
  int n;
  printf("Introduce el número: ");
  scanf("%d",&n);
  printf("El factorial es %d", factorial(n));

  return 0;
}
```

Por ejemplo el factorial de 4 tendría la siguiente secuencia de llamadas



5.2.2. Recursión lineal final

En la recursión lineal final el resultado que es devuelto es el resultado de ejecución de la última llamada recursiva. Un ejemplo de este cálculo es el máximo común divisor, que puede hallarse a partir de la fórmula:

$$\textbf{mcd(n,m)} \begin{cases} n & n = m \\ mcd(n-m,m) & n > m \\ mcd(n,m-n) & n < m \end{cases}$$

```
/* Calculo del maximo comun divisor con recursión */

#include <stdio.h>
long mcd(long,long);

main(int argc, char *argv[])
{
  long a= 4454,b= 143052;
  printf("El m.c.d. de %ld y %ld es %ld\n",a,b,mcd(a,b));
}

long mcd(long a, long b){
 if (a==b) return a;
 else if (a<b) return mcd(a,b-a);
 else return mcd(a-b,b);
}
```

5.2.3. Recursión múltiple

Alguna llamada recursiva puede generar más de una llamada a la función, como por ejemplo la serie de Fibonacci que se calcula mediante la fórmula:

$$\mathbf{F(n)} \begin{cases} 1 & \text{si } n = 1 \\ 1 & \text{si } n = 2 \\ F(n-1) + F(n-2) & \text{si } n > 2 \end{cases}$$

Estos números poseen múltiples propiedades, algunas de las cuales todavía siguen descubriéndose hoy en día, entre las cuales están:

1. La razón (el cociente) entre un término y el inmediatamente anterior varía continuamente, pero se estabiliza en un número irracional conocido como razón áurea o número áureo, que es la solución positiva de la ecuación $x^2 - x - 1 = 0$, y se puede aproximar por $1,618033989$.

2. Cualquier número natural se puede escribir mediante la suma de un número limitado de términos de la sucesión de Fibonacci, cada uno de ellos distinto a los demás. Por ejemplo, $17 = 13 + 3 + 1$, $65 = 55 + 8 + 2$.

3. Tan sólo un término de cada tres es par, uno de cada cuatro es múltiplo de 3, uno de cada cinco es múltiplo de 5, etc. Esto se puede generalizar, de forma que la sucesión de Fibonacci es periódica en las congruencias módulo m, para cualquier m.

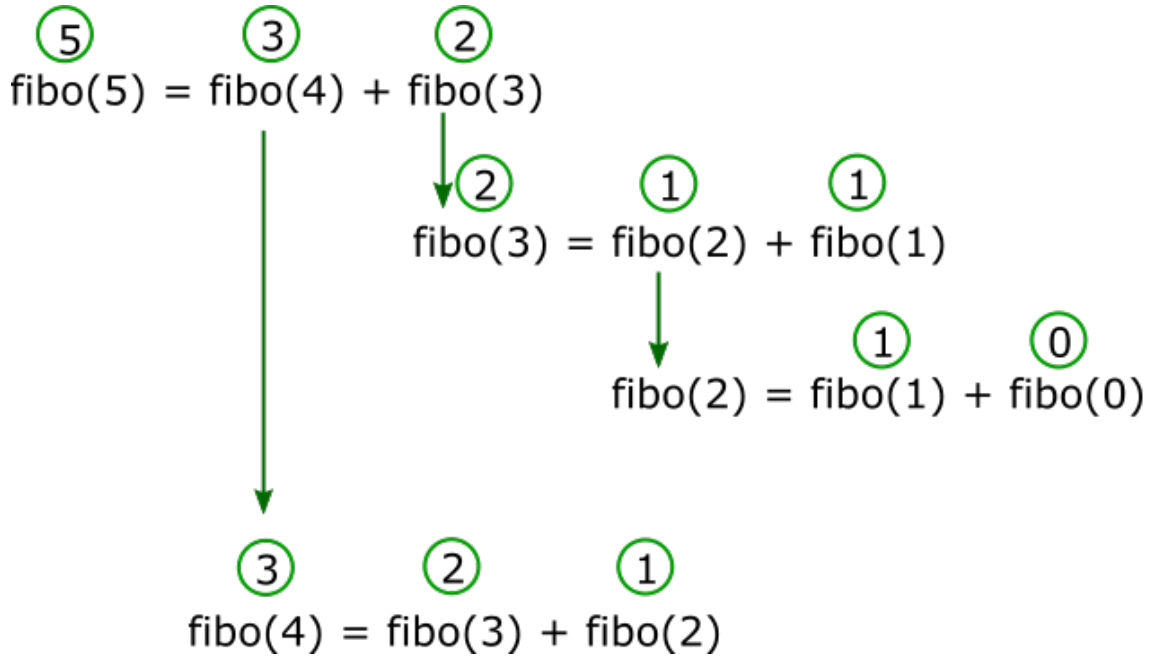
4. Si $F(p)$ es un número primo, p también es primo, con una única excepción. $F(4) = 3$; 3 es primo, pero 4 no lo es.

5. La suma infinita de los términos de la sucesión $F(n)/10^n$ es exactamente $10/89$.

Veamos un programa que nos permite calcular el número n de la serie de Fibonacci:

```
/*Fibo.c calculo de la serie de Fibonacci*/

#include<stdio.h>
int fibo(int n);
int main()
{
    int n, coef, i;
    printf("Ingrese la cantidad de terminos que desea ver de la serie de Fibonacci.\n");
    scanf("%d", &n);
    printf("Los coeficiente de la serie de Fibonacci son: \n");
```

```
    for(i=0; i<=n; i++)
    {
        coef = fibo(i);
        printf("coeficiente %d\t\t valor Fibonacci: %d \n ", i, coef);
    }
    printf("\n");


    return 0;
}

int fibo(int n)
{
    if (n>1)
    {
        return fibo(n-1)+fibo(n-2);
    }

    else
    {
        if(n==0)
            return 0;
        if(n==1)
            return 1;
    }

}
```



5.2.4. Recursión mutua

Implica más de una función que se llaman mutuamente. Un ejemplo es el determinar si un número es par o impar mediante dos funciones:

```
/*  Determinar si un numero es par o impar mediante recursión    */
```

```
#include <stdio.h>

void main()
{
 int n= 29;
 if (par(n))
 printf("El numero es par");
 else
 printf("El numero es impar");
}

int par(int n){
 if (n==0) return 1;
 else return (impar(n-1));
}
int impar(int n){
 if (n==0) return 0;
  else return(par(n-1));
}
```

5.3. Problemas recursión

1. Construir una función recursiva que calcule la suma de los n primeros números naturales.
2. Construir una función recursiva que imprima la lista de números naturales comprendidos entre dos valores a y d dados por el usuario.
3. Escribir una función recursiva que devuelva la cantidad de dígitos de un número entero.
4. Escribir una función recursiva que calcule x^y mediante multiplicaciones sucesivas, siendo x e y dos números enteros.
5. Escribir una función recursiva que calcule $x * y$ mediante sumas sucesivas, siendo x e y dos números enteros.
6. Calcula mediante un diseño recursivo el valor máximo de un vector de componentes numéricas.
7. Construir una función recursiva que cuente el número de secuencias de dos 1 seguidos que hay en una cadena de caracteres que represente un número binario.
8. Escribir la función recursiva que recibiendo como parámetros una cadena de dígitos hexadecimales y su longitud devuelva el valor decimal que representa dicha cadena.
9. Modificar el programa anterior para que la secuencia de 1 sea de longitud m.
10. Calcular $C(n,k)$ siendo:
$$C(n,0) = C(n,n) = 1 sin >= 0$$
$$C(n,k) = C(n-1,k) + C(n-1,k-1) sin > k > 0$$
11. Escriba un programa que implemente una función que reciba una cadena de caracteres y en forma recursiva devuelva la cadena invertida.
12. Implemente una función recursiva que nos diga si una cadena es palíndromo.
13. Implemente una función recursiva que tomado un vector de enteros devuelva la suma de sus elementos.

14. Implemente una función recursiva que tomado un vector de enteros devuelva la suma cuadratica de la diferencia de sus elementos con respecto a la media.