Programación de Computadoras

Dr. Félix Calderon Solorio

27 de junio de 2013

Índice general

Introducción al Lenguaje C

1.1. Introducción a los Lenguajes de Programación

Algunas estructuras básicas que podemos ver en la mayoría de los lenguajes son: condicionales, sentencias de repetición y funciones. A continuación haremos una definición de las mismas que nos ayudaran a entender como realizar un programa

Una sentencia condicional la podemos escribir como

Si (condición se cumple) **entonces**
{
realizar tarea1
}
Si no
{
realizar tarea2
}

Una sentencia de repetición, la cual esta sujeta al cumplimiento de una condición especifica es

Mientras (condición se cumple)
{
realizar tarea2
}

Esta repetición también podría llevarse de la siguiente manera.

Repetir
{
realizar tarea2
} **Mientras** (condición se cumple)

A diferencia de la anterior este bloque se ejecutar· al menos una sola vez, ya que primero se realiza la tarea y después se pregunta por la condición.

Otra manera de implementar una sentencia de repetición es haciendo una estructura que repita, un número de veces fijo, una tarea especifica así

Para (x = 1, 2, 3, ... 10)
{
realizar tarea
}

Finalmente podemos hacer conjuntos de estas funciones de tal forma que realicen una tarea especifica. Este tipo de sentencias es una función la cual podemos definir como

nombre (variables necesarias)
{
grupo de sentencias 1
grupo de sentencias 2
...
}

1.1.1. Ejemplo

Hacer una función que sumar los números del 1 al 10. El algoritmo resultante se presenta en (algoritmo 1).

Algoritmo 1 Sumador

Entrada:
Salida: *suma*
  $x = 1$
  $suma = 0$
  mientras $(x < 11)$ **hacer**
    $suma \leftarrow suma + x$
    $x \leftarrow x + 1$
  fin mientras

1.1.2. Ejemplo

Hacer una función que sume los números pares de a a b.

Algoritmo 2 Sumador Par

Entrada: a, b
Salida: $suma$
  $x \leftarrow a$
  mientras $(x <= b)$ **hacer**
    si Es_par (x) **entonces**
      $suma \leftarrow suma + x$
      $x \leftarrow x + 1$
    fin si
  fin mientras

1.2. Características del lenguaje C

C es un lenguaje estructurado de nivel medio, ni de bajo nivel como ensamblador, ni de alto nivel. Esto permite una mayor flexibilidad y potencia, a cambio de menor abstracción.

No se trata de un lenguaje fuertemente tipado, lo que significa que se permite casi cualquier conversión de tipos. No es necesario que los tipos sean exactamente iguales para poder hacer conversiones, basta con que sean parecidos.

No lleva a cabo comprobación de errores en tiempo de ejecución, por ejemplo no se comprueba que no se sobrepasen los límites de los arreglos. El programador es el único responsable de llevar a cabo esas comprobaciones.

Tiene un reducido numero de palabras clave, unas 40.

auto	*break*	*case*	*char*	*const*	*continue*	*default*
do	*double*	*else*	*enum*	*extern*	*float*	*for*
goto	*if*	*int*	*long*	*register*	*return*	*short*
signed	*sizeof*	*static*	*struct*	*typedef*	*union*	*unsigned*
void	*volatile*	*while*				

El lenguaje C dispone de una biblioteca estándar que contiene numerosas funciones y que siempre está disponible, además de las extensiones que proporcione cada compilador o entorno de desarrollo. En resumen, es un lenguaje muy flexible, muy potente, muy popular, pero que no protege al programador de sus errores.

1.3. Compilación de archivos fuente

1.3.1. Compilación

Para ejecutar un código desarrollado en cualquier lenguaje de programación ser· necesario realizarlo en dos pasos. Primeramente deber· ser traducido el código original, al que llamaremos fuente, a un código binario, que finalmente es el único lenguaje que entiende la computadora. después de realizado esto, el código podrá ser ejecutado y presentar los resultados.

La manera de realizar este procedimiento cambia de lenguaje a Lenguaje. En seguida se presenta el proceso en C

El proceso de compilación crear· un código, que llamaremos objeto, el cual toma el mismo nombre que la clase principal puesta en el programa fuente y crea un archivo nombre.class.

compilación y ejecución de un programa en C

Dado un código fuente, que generalmente tendrá extensión .c (nombre.c), el proceso de compilación se realiza como

gcc -o nombre nombre.c

y la ejecución es

./nombre

Donde -o indica al compilador el nombre del archivo ejecutable, si este es omitido, en su lugar se generar· un archivo a.out.

En el caso de tener mas de un archivo, en común que se creen proyectos los cuales se compilan con la instrucción make y una archivo Makefile debe ser creado. Dentro del archivo Makefile se especifican los archivos fuente y el nombre del archivo de salida.

1.3.2. Ejemplo del programa Hola Mundo

A continuación se da un ejemplo sencillo para desplegar un letrero utilizando el lenguaje C.

```
//ejemplo ec001.cpp
/
 * Title: Hola Mundo
```

```
 * Description: Programa para desplegar el letrero de Hola Mundo
 * Copyright: Copyright (c) 2005
 * Company: UMSNH
 * author Dr. Felix Calderon Solorio
 * version 1.0
 */
#include <stdio.h>

void main () {
  printf("Hola Mundo \n");
  return;
}
```

Ver Hola_Mundo.c

Podemos notar lo siguiente:

- Que se requiere de una función principal la cual se denomina main
- Que en C es siempre necesario incluir librerías utilizando la palabra clave

  ```
  #include <stdio.h>
  ```

- Que la función main son opcionales los argumentos.

1.3.3. Iniciales

El siguiente programa escribe las iniciales FCS utilizando solamente la función printf que permite imprimir un texto en pantalla. Estas iniciales estas realizadas utilizando asteriscos para que las iniciales luzcan en un tamaño mayor.

```
/**
 * El metodo imprime las iniciales FCS en orden vertical
 */

#include <stdio.h>

int main ()
{
    printf("********\n");
    printf("    *  *\n");
    printf("    *  *\n");
    printf("    *  *\n");
```

```
    printf("\n");
    printf("\n");
    printf("\n");
    printf("********\n");
    printf("*      *\n");
    printf("*      *\n");
    printf("*      *\n");
    printf("\n");
    printf("\n");
    printf("\n");
    printf("*   ****\n");
    printf("*   *  *\n");
    printf("*   *  *\n");
    printf("*****  *\n");
    printf("\n");
    printf("\n");

    return 0;
}
```

Ver Iniciales.c

Tipos de datos, constantes, operadores y expresiones

2.1. Nombre de variables, tipos de datos

En nuestro primer ejemplo escribíamos un texto en pantalla, pero este texto estaba prefijado dentro de nuestro programa. Esto no es lo habitual: normalmente los datos que maneje nuestro programa serán el resultado de alguna operación matemática o de cualquier otro tipo. Por eso, necesitaremos un espacio en el que ir almacenando valores temporales y resultados.

En casi cualquier lenguaje de programación podremos reservar esos "espacios", y asignarles un nombre con el que acceder a ellos. Esto es lo que se conoce como "variables".

Por ejemplo, si queremos que el usuario introduzca dos números y el ordenador calcule la suma de ambos números y la muestre en pantalla, necesitaríamos el espacio para almacenar al menos esos dos números iniciales (en principio, para la suma no sería imprescindible, porque podemos mostrarla en pantalla nada más calcularla, sin almacenarla previamente en ningún sitio). Los pasos a dar serían los siguientes:

- Pedir al usuario que introduzca un número.
- Almacenar ese valor que introduzca el usuario (por ejemplo, en un espacio de memoria al que podríamos asignar el nombre "primerNumero").
- Pedir al usuario que introduzca otro número.
- Almacenar el nuevo valor (por ejemplo, en otro espacio de memoria llamado segundoNumero)
- Mostrar en pantalla el resultado de sumar primerNumero y segundoNumero.

Pues bien, en este programa estaríamos empleando dos variables llamadas primerNumero y segundoNumero. Cada una de ellas sería un espacio de memoria capaz de almacenar un

número.

Los nombres de las variables en C siempre comienzan con una letra del alfabeto o el carácter _, seguido de letras, números y el mismos carácter _. Así tendremos que nombres validos de variables son:

`Temperatura Calor Tiempo_1 Tiempo_2`

El espacio de memoria que hace falta "reservar" será distinto según la precisión que necesitemos para ese número, o según lo grande que pueda llegar a ser dicho número. Por eso, tenemos disponibles diferentes "tipos de variables".

Por ejemplo, si vamos a manejar números sin decimales ("números enteros") de como máximo 9 cifras, nos interesaría el tipo llamado int (abreviatura de integer, que es entero en inglés). Este tipo de datos consume un espacio de memoria de 4 bytes (1 Mb es cerca de 1 millón de bytes). Si no necesitamos guardar datos tan grandes (por ejemplo, si nuestros datos va a ser números inferiores a 1.000), podemos emplear el tipo de datos llamado short (entero corto).

Los tipos de datos numéricos disponibles en C son los siguientes:

Nombre	Admite decimales	Valor min	Valor max	Presición	ocupa
char	no	-128	127	-	1 byte
unsigned char	no	0	255	-	1 byte
short int	no	-32768	32767	-	2 bytes
int	no	-2147483648	2147483647	-	4 bytes
long int	no	-9223372036854775808	9223372036854775807	-	8 bytes
float	sí	1.401298E-45	3.402823E38	6-7 cifras	4 bytes
double	sí	4.94065645841247E-324	1.79769313486232E308	14-15 cifras	8 bytes

La forma de declarar variables de estos tipos es la siguiente:

```
int numeroEntero;    // La variable numeroEntero será un número de
tipo "int" short distancia;     // La variable distancia guardará
números "short" long gastos;          // La variable gastos es de
tipo "long" byte edad;            // Un entero de valores
"pequeños" float porcentaje;    // Con decimales, unas 6 cifras de
precisión double numPrecision; // Con decimales y precisión de unas 14 cifras
```

(es decir, primero se indica el tipo de datos que guardará la variable y después el nombre que queremos dar a la variable).

Se pueden declarar varias variables del mismo tipo a la vez:

```
int primerNumero, segundoNumero;    // Dos enteros
```

y también se puede dar un valor a las variables a la vez que se declaran:

```
int a = 5;            // "a" es un entero, e inicialmente vale 5
short b=-1, c, d=4;   // "b" vale -1, "c" vale 0, "d" vale 4
```

Los nombres de variables pueden contener letras y números (y algún otro símbolo, como el de subrayado, pero no podrán contener otros muchos símbolos, como los de las distintas operaciones matemáticas posibles, ni podrán tener vocales acentuadas o eses).

Un ejemplo sencillo de declaración de tipos es:

```
/*
 * File:   tipos.c
 * Author: felix
 *
 * Created on 22 de agosto de 2011, 19:36
 */


/*
 * Este programa muestra ejemplo utilizando los diferentes tipos de datos
 * basicos en C
 */
# include <stdio.h>

int main() {

    // Tipo de dato Entero

    int a = 1;
    unsigned int b = 10;
    signed long c = 5;
    signed long long d = 10000000;

    printf("a = %d \n", a);

    // Tipo de dato Flotante

    float e;
    double f = 1e23;
    double g = 3.1416;
    float h = 4e-9;
    double i = -78;

    printf("%e en flotante es %10.10f en entero es %d\n", h, h, h);

    // tipo de dato caracter
```

```
    char j;
    char k = 's';
    char l = 48;

    printf("%c = %d en ascii \n", k, k);

    return (0);
}
```

Ver tipos.c

Vamos a aplicarlo con un ejemplo sencillo en C que sume dos números:

```
/*
 * File:   suma2.c
 * Author: felix
 *
 * Created on 13 de febrero de 2012, 12:18
 */

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {

     int primerNumero = 56;    // Dos enteros con valores prefijados
     int segundoNumero = 23;

     printf( "La suma es: %d\n" , primerNumero+segundoNumero );

    return (EXIT_SUCCESS);
}
```

Hay una importante diferencia entre las dos ordenes printf: la primera contiene comillas, para indicar que ese texto debe aparecer tal cual, mientras que la segunda no contiene comillas, por lo que no se escribe el texto primerNumero + segundoNumero, sino que se intenta calcular el valor de esa expresión (en este caso, la suma de los valores que en ese momento almacenan las variables primerNumero y segundoNumero, 56+23 = 89).

Lo correcto habría sido que los datos no los hubiéramos escrito nosotros en el programa, sino que fuera el usuario quien los hubiera introducido, pero esto es algo relativamente complejo

cuando creamos programas en modo texto en C (al menos, comparado con otros lenguajes anteriores), y todavía no tenemos los conocimientos suficientes para crear programas en modo gráfico, así que más adelante veremos cómo podría ser el usuario de nuestro programa el que introdujese los valores que desea sumar.

Tenemos otros dos tipos básicos de variables, que no son para datos numéricos: char. Será una letra del alfabeto, o un dígito numérico, o un símbolo de puntuación y ocupa 1 byte. Estos ocho tipos de datos son lo que se conoce como tipos de datos primitivos (porque forman parte del lenguaje estándar, y a partir de ellos podremos crear otros más complejos). El código ASCCI se puede ver en la tabla 2.1

2.2. Operadores y expresiones; aritméticos, lógicos, a nivel de bits y de asignación

2.2.1. Operaciones aritméticas básicas

Hay varias operaciones matemáticas que son frecuentes. Veremos cómo expresarlas en C, y también veremos otras operaciones menos frecuentes. Las más comunes son:

Operación matemática	Símbolo correspondiente
Suma	+
Resta	-
Multiplicación	*
División	/

El siguiente código muestra una implementación para el manejo de operaciones.

```
/*
 * File:   expresiones_aritmeticas.c
 * Author: felix
 *
 * Created on 22 de agosto de 2011, 19:48
 */

#include <stdio.h>
#include <stdlib.h>

/*
 *
 */
int main() {
```

Cuadro 2.1: Códigos ASCII

Dec	Hex	Rep	Dec	Hex	Rep	Dec	Hec	Rep
32	20	esp	64	40	@	96	60	‘
33	21	!	65	41	A	97	61	a
34	22		66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	’	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[	123	7B	{
60	3C	¡	92	5C		124	7C	—
61	3D	=	93	5D	]	125	7D	}
62	3E	¿	94	5E		126	7E	
63	3F	?	95	5F				

```
    float a = 10, b = 30, c;

    // Suma

    c = a + b;

    printf("La suma de %f y %f es %f\n", a, b, c);

    // Resta

    c = a - b;

    printf("La resta de %f y %f es %f\n", a, b, c);

    // Multiplicacion

    c = a * b;

    printf("La multiplicacion de %f y %f es %f\n", a, b, c);

    // Modulo. Calcula el reciduo de la division entera entre dos numeros

    int d = 10, e = 3, f;

    f = d % e;

    printf("El residuo de dividir %d entre %d es %d\n", d, e, f);

    return (0);
}
```

Hemos visto un ejemplo de cómo calcular la suma de dos números; las otras operaciones se emplearían de forma similar y en el ejemplo anterior se muestra.

Otras operaciones matemáticas menos habituales. Esas son las operaciones que todo el mundo conoce, aunque no haya manejado computadoras. Pero hay otras operaciones que son frecuentes en los lenguajes de programación y no tanto para quien no ha manejado computadoras, pero que aun así deberíamos ver:

Operación	Símbolo
Resto de la división	%

El resto de la división también es una operación conocida. Por ejemplo, si dividimos 14 entre 3 obtenemos 4 como cociente y 2 como resto, de modo que el resultado de 14 % 3 sería 2. El siguiente ejemplo muestra como se aplica la operación de modulo o resto de la división y su diferencia con la división. Hay que recordar que la operación de modulo solamente se aplica en números enteros.

```
/*
 * File:   division.c
 * Author: felix
 *
 * Created on 30 de agosto de 2011, 13:59
 */

#include <stdio.h>
#include <stdlib.h>

int main() {

    int Dividendo, Divisor, Cociente, Residuo;

    Dividendo = 120;
    Divisor = 33;
    Cociente = Dividendo/Divisor;
    Residuo = Dividendo%Divisor;

    printf("El resultado de dividir %d entre %d es %d y sobra %d\n",
            Dividendo, Divisor, Cociente, Residuo);

    return (0);
}
```

Otra aplicación del modulo o resto de la división es el calculo de múltiplos de un número, así por ejemplo cuando un numero n es múltiplo de m el resultado de la operación es cero

$$n\,\%m = 0$$

2.2.2. Incremento y asignaciones abreviadas

Hay varias operaciones muy habituales, que tienen una sintaxis abreviada en C. Por ejemplo, para sumar 2 a una variable a, la forma normal de conseguirlo sería:

a = a + 2;

pero existe una forma abreviada :

a += 2;

Al igual que tenemos el operador += para aumentar el valor de una variable, tenemos -= para disminuirlo, /= para dividirla entre un cierto número, *= para multiplicarla por un número, y así sucesivamente. Por ejemplo, para multiplicar por 10 el valor de la variable "b" haríamos

b *= 10;

También podemos aumentar o disminuir en una unidad el valor de una variable, empleando los operadores de "incremento" (++) y de decremento (–). Así, para sumar 1 al valor de **a**, podemos emplear cualquiera de estas tres formas:

a = a +1; a += 1; a++;

Los operadores de incremento y de decremento se pueden escribir antes o después de la variable. Así, es lo mismo escribir estas dos líneas:

a++; ++a;

Pero hay una diferencia si ese valor se asigna a otra variable al mismo tiempo que se incrementa/decrementa:

int c = 5; int b = c++;

da como resultado c = 6 y b = 5 (se asigna el valor a b antes de incrementar c) mientras que

int c = 5; int b = ++c;

da como resultado c = 6 y b = 6 (se asigna el valor a b después de incrementar c).

Como ejemplo de aplicación tenemos el siguiente código en C, donde se aplican algunas de las expresiones de incremento.

```
/*
 * File:   preincremento_postincremento.c
 * Author: felix
 *
 * Created on 23 de agosto de 2011, 8:28
 */

#include <stdio.h>

int main() {
    int c;
```

```
    c=5;
    printf("Post incremento\n");
    printf("%d\n", c);
    printf("%d\n", c++);
    printf("%d\n", c);

    c=5;

    printf("Pre incremento\n");
    printf("%d\n", c);
    printf("%d\n", ++c);
    printf("%d\n", c);

    c=5;
    printf("Post decremento\n");
    printf("%d\n", c);
    printf("%d\n", c--);
    printf("%d\n", c);

    c=5;

    printf("Pre decremento\n");
    printf("%d\n", c);
    printf("%d\n", --c);
    printf("%d\n", c);

    return (0);
}
```

2.2.3. Operaciones a nivel de bits

Las operaciones a nivel de bits deberían ser habituales para los estudiantes de ciencias computacionales, pero quizás no tanto para quien no haya trabajado con el sistema binario. No entraré por ahora en más detalles, salvo poner un par de ejemplos para quien necesite emplear estas operaciones: para desplazar los bits de a dos posiciones hacia la izquierda, usaríamos $a << 2$; para invertir los bits de c (complemento) usaríamos $\tilde{}c$; para hacer una suma lógica de los bits de d y f sería $d|f$. Estas operaciones son muy utilizadas por estudiantes de electrónica para poder realizar operaciones a nivel de lenguaje de máquina.

Operación	Símbolo
Desplazamiento de bits a la derecha	>>
Desplazamiento de bits a la derecha	<<
Desplazamiento rellenando con ceros	>>>
Producto lógico (and)	&
Suma lógica (or)	|
Suma exclusiva (xor)	^
Complemento	~

Para entender las operaciones and, or, exor y not es importante hacer uso de las tablas de verdad como se muestra enseguida.

And

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

Or

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

Xor

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

Not

x	~ y
0	1
1	0

El siguiente programa es un ejemplo en C de como se utilizan las expresiones a nivel de bits

```
/*
 * File:   Expresiones_bits.c
 * Author: felix
 *
 * Created on 13 de septiembre de 2011, 13:49
 */
```

```
#include <stdio.h>
#include <stdlib.h>

int main() {

    int mask = 0xFF;
    int a = 10, b = 8;

    // Operacion And
    printf("AND a & b = %X\n", a&b);
    // Operacion Or
    printf("OR  a | b = %X\n", a|b);
    // Operacion Xor
    printf("XOR a ^ b = %X\n", a^b);
    // Operacion de complemento
    printf("com   ~a = %X\n", (~a));

    printf("El entero mas grades es %u\n", ~0);

    return (0);
}
```

2.2.4. Operadores relacionales

También podremos comprobar si entre dos números (o entre dos variables) existe una cierta relación del tipo ¿es a mayor que b? o ¿tiene a el mismo valor que b?. Los operadores que utilizaremos para ello son:

Operación	Símbolo
Mayor que	$>$
Mayor o igual que	$>=$
Menor que	$<$
Menor o igual que	$<=$
Igual que	$==$
Distinto de	$!=$

Estas operaciones dan como resultado Cierto o Falso que en general se conoce como tipo Booleano. Sin embargo en C no existe tal tipo de datos y son remplazados por dos valores numéricos que son 0 para falso y 1 para verdadero. Esto da una posibilidad única a C, de tratar de manera indistinta expresiones lógicas y numéricas.

así, por ejemplo, para ver si el valor de una variable b es distinto de 5, escribiríamos algo

parecido (veremos la sintaxis correcta un poco más adelante) a

```
SI  b != 5 ENTONCES ...
```

o para ver si la variable a vale 70, sería algo como (nuevamente, veremos la sintaxis correcta un poco más adelante)

```
SI  a == 70 ENTONCES ...
```

Es muy importante recordar esta diferencia: para asignar un valor a una variable se emplea un único signo de igual, mientras que para comparar si una variable es igual a otra (o a un cierto valor) se emplean dos signos de igual.

Operadores lógicos. Podremos enlazar varias condiciones, para indicar qué hacer cuando se cumplan ambas, o sólo una de ellas, o cuando una no se cumpla. Los operadores que nos permitirán enlazar condiciones son:

Operación	Símbolo
Y	&&
O	||
No	!

Por ejemplo, la forma de decir si a vale 3 y b es mayor que 5, o bien a vale 7 y b no es menor que 4 en una sintaxis parecida a la de C (aunque todavía no es la correcta) sería:

```
SI (a==3 && bb>5) || (a==7  && ! (b<4))
```

El siguiente es un ejemplo de como luce la implementación en C de las expresiones lógicas.

```
/*
 * File:   Expresiones_logicas.c
 * Author: felix
 *
 * Created on 22 de agosto de 2011, 19:55
 */

#include <stdio.h>
#include <stdlib.h>

/*
 * Operadores Logicos
 */
int main() {

    // == operador de igualdad en C
```

```
    // != operador de desigualdad
    // > mayor
    // < menor
    // >= mayor o igual
    // <= menor o igual

    // && ( Y logico)
    // || (O logico)
    // ! (NO logico o negacion logica)

    int a = 10, b =30, c=5, d;

    printf("1 es cierto \n0 es falso\n\n");

    // And logico

    d = (a>b) && (b>c);

    printf("%d\n", d);

    // Or logico

    d = (a>b) || (b>c);

    printf("%d\n",  d);

    return (0);
}
```

2.2.5. Ejemplo

Indique en que casos las siguientes sentencias son verdaderas o falsas dados los valores a=1 b = 2 y c = 5;

```
1.- d = (a*b) < (c-4);

2.- d = (a < b) && (c!=4);

3.- d = (a &b)>2;

4.- d = ((c *c) | 5) <=4;
```

Las respuestas son false, true, false y false.

En virtud de que el resultado de las operaciones lógicas es 1 o 0, estos valores pueden ser introducidos dentro de expresiones aritméticas y representar un tipo de funciones a las que se llama discontinuas. Así por ejemplo la siguiente función es llamada función a pedazos y su definición es:

$$f(x) = \begin{cases} si & x > 0 & x^2 \\ si & no & x \end{cases}$$

La implementación se muestra en el siguiente código donde además se hace una aplicación para determinar si una letra es mayúscula o minúscula.

```
/*
 * File:   funcion_pedazos.c
 * Author: felix
 *
 * Created on 6 de septiembre de 2011, 12:46
 */

#include <stdio.h>
#include <stdlib.h>

int main() {

    double x = -3;

    double f = (x>=0)*x*x + (x<0)*x;
    printf("f(%f) = %f\n", x, f);

    x = 5;
    f = (x>=0)*x*x + (x<0)*x;
    printf("f(%f) = %f\n", x, f);

    printf("\n\n\n");
    char letra = 'A';
    char mayus = 's'*(letra>= 'A' && letra <='Z') + 'n'*(letra>= 'a' && letra <='z') ;
    printf("La letra %c es mayuscula %c\n", letra, mayus);

    letra = 'c';
    mayus = 's'*(letra>= 'A' && letra <='Z') + 'n'*(letra>= 'a' && letra <='z') ;
    printf("La letra %c es mayuscula %c\n", letra, mayus);
```

```
    return (0);
}
```

2.3. Precedencia de los operadores

Las reglas de precedencia de operadores son una guía de acción, que le permiten a los lenguajes calcular expresiones en el orden correcto. La siguiente tabla presenta el orden de precedencia

Operadores	Operaciones	Orden de cálculo
()	Paréntesis	Se calculan primero.
$*, /, \%$	Multiplicación, División y Modulo	Se evalúan en segundo lugar
$+, -$	Suma o resta	Se calcula al último

Ejercicio

Dada la ecuación $y = ax^3+7$, ¿cual de los que siguen, si es que existe alguno, son enunciados correctos correspondientes a esta ecuación?.

```
a) y = a * x * x * x + 7;

b) y = a * x * x * (x + 7);

c) y = (a * x) * x * (x + 7);

d) y = (a * x) * x * x + 7;

e) y = a * (x * x * x) + 7;

f) y = a * x * (x * x + 7);
```

Ejercicio

Declare el orden de cálculo de los operadores en cada uno de los enunciados siguientes y muestre el valor de x una vez que se ejecute cada una de ellas.

```
a) x = 7 + 3 * 6 / 2 - 1;

b) x = 2 \% 2 + 2 * 2 - 2 / 2;

c) y = (3 * 9 * (3 + (9 * 3 / (3))));
```

2.3.1. Literales, Constantes y variable

Con el objetivo de generalizar nuestros programa haremos uso de letras que representarán los valores que deseamos guardar para un procesamiento posterior en tiempo de ejecución. Las variables podrán ser de cualquiera de los tipos numéricos, de tipo carácter o los que tipos compuestos definidos por los usuarios.

Los nombres de las variables validos comienzan con una letra seguida de otras letras o el carácter _. La longitud de los nombres no tiene limite y normalmente se sugiere, por facilidad de lectura, tengan relación con la variable que representa.

así por ejemplo
`int a= 10;`
almacenara el número 10 en la variable a.

`char a = 'a'`
almacenara en la variable a el carácter a. Note que los caracteres se pondrán entre comillas a diferencia de los nombres de variables.

2.4. Cálculo del valor máximo de un tipo, overflow

Dado que los tipos de datos tienen límites, debemos ser cuidadosos al realizar las operaciones ya que ocurrirán cosas raras. Un ejemplo es el caso de una variable de tipo char a la cual se le asigna el valor de 127 y se le suma un 1 el resultado no es 128 sino -128. Esto se debe a que ocurre un sobre flujo (overflow) y el número siguiente que se asigna no es el consecutivo que esperamos sino el del límite inferior.

El siguiente código muestra algunas operaciones que producen sobre flujo en los datos y para evitarlo debemos siempre tener en cuenta los rangos en los cuales las variables están definidas.

```
/*
 * File:   rango_tipos.c
```

```
 * Author: felix
 * Operaciones para indicar el rango de los tipos de datos
 * Created on 7 de septiembre de 2011, 17:41
 */

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    /* Tipo de dato caracter */

    char a = 127+1;
    printf("a = %d\n", a);

    unsigned char b = 127 +1;
    printf("b = %d\n", b);

    unsigned char c = 255 +1;
    printf("c = %d\n", c);

    /* Tipo de dato entero */

    unsigned int y = 0;
    printf("y = %u\n", y-1);

    int d = 2147483648 +1;
    printf("d = %d\n", d);

    unsigned e = 2*d;
    printf("e = %d\n", e);

    long f = d+1;
    printf("f = %d\n", f);

    unsigned long g = 2*d;
    printf("g = %d\n", g);


    int x = 10 >> 2;
    printf("%d\n", x);
```

```
    return (EXIT_SUCCESS);
}
```

2.5. Instrucciones secuenciales

Una manera de realizar un sencillo programa es utilizando una serie de instrucciones de forma consecutiva (secuencial o en secuencia) que permita realizar una tarea. Algunos ejemplos de ellos se presentan en las siguientes tareas.

2.6. Ejemplos

2.6.1. Calculo de áreas y volúmenes de figuras geométricas regulares

Dada la siguiente información, hacer un programa que permita calcular el área de un triángulo, de un cuadrado, y de un rectángulo, así como los volúmenes de un cilindro y de un cono.

El triángulo es un polígono formado por tres lados y tres ángulos. La suma de todos sus ángulos siempre es 180 grados. Para calcular el área se emplea la siguiente fórmula:

Área del triángulo = (base * altura) / 2

El cuadrado es un polígono de cuatro lados, con la particularidad de que todos ellos son iguales. Además sus cuatro ángulos son de 90 grados cada uno. El área de esta figura se calcula mediante la fórmula:

Área del cuadrado = lado al cuadrado

El rectángulo es un polígono de cuatro lados, iguales dos a dos. Sus cuatro ángulos son de 90 grados cada uno. El área de esta figura se calcula mediante la fórmula:

Área del rectángulo = base * altura

El cilindro es el sólido engendrado por un rectángulo al girar en torno a uno de sus lados. Para calcular su área lateral, su área total así como para ver su desarrollo pulsar sobre la figura anterior Para calcular su volumen se emplea la siguiente fórmula:

Volumen del cilindro = área de la base * altura

El cono es el sólido engendrado por un triángulo rectángulo al girar en torno a uno de sus catetos. Para calcular su área lateral, su área total así como para ver su desarrollo pulsar sobre la figura anterior Para calcular su volumen se emplea la siguiente fórmula:

Volumen del cono = (área de la base * altura) / 3

La implementación en C es:

```
// Ejemplo ec021.cpp
include <stdio.h>

/**
 * Title: Calculo de areas y volumenes
 * Calcula utilizando simples sentencias las áreas y volúmenes de figuras
 * 2005
 * Company: UMSNH
 * @author Dr. Felix Calderon Solorio
 * @version 1.0
 */

int main()
 {

    // datos

  double base = 10, altura = 5, radio = 2;

  // areas

  double Atriangulo  = (base * altura) / 2.0;
  double Acuadrado   = base*base;
  double Arectangulo = base * altura;

  // volumenes

  double Abase       = 3.1416*radio*radio;
  double Vcilindro   = Abase * altura ;
  double Vcono       = Abase * altura / 3.0;

  printf("Area de triangulo  = %f\n", Atriangulo);
  printf("Area de cuadrado   = %f\n", Acuadrado);
  printf("Area de rectangulo = %f\n", Arectangulo);
  printf("Volumen cilindro   = %f\n", Vcilindro);
  printf("Volumen cono       = %f\n", Vcono);

  return 0;
```

```
}
```

2.6.2. Ley de ohm y Análisis básico de transistores

Para el circuito mostrado en la figura 2.1, determinar la ganancia de voltaje de un transistor en configuración de base común.

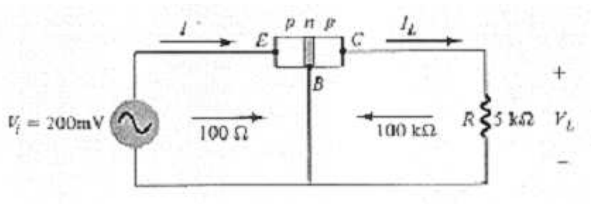


Figura 2.1: Ejemplo de transistores.

La Solución en C queda como:

```
// Ejemplo ec023.cpp

#include <stdio.h>

/**
 * Title: Calculo de la ganancia de voltaje para un transistor
 * Description: Programa para determinar ganancias de voltaje
 * Copyright: Copyright (c) 2005
 * Company: UMSNH
 * author Dr. Felix Calderon Solorio
 * version 1.0
 */

int main() {

  double Vi = 200e-3, Re = 100, Rc = 100e+3, R = 5e3;
  double I, Il, Vl, Av;

  // calculo de la corriente de emisor.

  I = Vi/Re;
  printf("I = %f\n", I);
```

```
  // Asumiendo una ganacia alpha = 1 y que Ic = Ie

  Il = I;
  printf("Il = %f\n", Il);

  // el voltaje en la resistencia R es

  Vl = Il*R;
  printf("Vl = %f\n", Vl);


  // finalmente la ganacia de voltaje es

  Av = Vl/Vi;
  printf("Av = %f\n", Av);

}
```

Entrada de datos por teclado y salida por consola

Una parte importante de la solución de cualquier problema es la presentación de resultados. Es este capítulo analizaremos a fondo las características de formato printf, scanf, getche y gets en C.

3.1. Entrada Salida en C

.

Las funciones de entrada salida se encuentran en la librería `#include stdio.h` y son un conjunto de funciones que nos permiten realizar entrada de datos por teclado, salida a pantalla y entrada y salida a archivos. En este capítulo solamente se analizaran algunas de las funciones de entrada desde teclado y salida a pantalla. Las funciones que tienen que ver con archivos se dejarán para capiítulos posteriores.

Algunas de las funciones de entrada salida son: printf, puts, scanf, gets, getchar, etc.

3.1.1. Función prinft

La función printf se utiliza para escribir una cadena de caracteres o bien un conjunto de números con formato. La sintaxis general de esta función es

```
printf("Cadena de formato de control", argumentos);
```

Los formatos numéricos de esta función se muestran a continuación.

Especificador	Descripción
d	Despliega un entero decimal con signo
i	Despliega un entero decimal con signo
o	Muestra un entero octal sin signo
u	Muestra un estero decimal sin signo
x o X	Muestra un entero en Hexadecimal x las letras aparecen con minúsculas X las letras aparecen con mayúsculas
h o l	Se coloca antes de cualquier especificador de conversión de enteros para indicar short o bien long, respectivamente

Un ejemplo del uso de esta función para el caso de enteros se muestra en la siguiente código

Ejemplo formatos_enteros.c

```
/**
  * Ejemplo para mostrar los diferentes formatos de
  * la función printf, con numeros enteros.
*/

#include <stdio.h>

int main() {
    printf("%d\n", 511);
    printf("%i\n", 511);
    printf("%d\n", +511);
    printf("%d\n", -511);
    printf("%hd\n", 32000);
    printf("%ld\n", 2000000000);
    printf("%o\n", 511);
    printf("%u\n", 511);
    printf("%u\n", -511);
    printf("%x\n", 511);
    printf("%X\n", 511);

    return 0;
}
```

Para imprimir números flotantes, la función printf, utiliza los siguientes especificadores

Especificador	Descripción
e o E	Muestra un valor en punto flotante en notación exponencial
f	Muestra valores de punto flotante
g o G	Despliega un valor en punto flotante, ya sea en la forma forma de punto flotante f o en la forma exponencial e o E
L	Indicador de flotante largo long double

```
// Ejemplo formatos_flotantes.c

/**
  * Ejemplo para mostrar los diferentes formatos de
  * la función printf, con numeros flotantes.
*/

#include <stdio.h>

int main() {
    printf("%e\n", 1234567.89);
    printf("%e\n", +1234567.89);
    printf("%E\n", -1234567.89);
    printf("%f\n", 1234567.89);
    printf("%g\n", 1234567.89);
    printf("%G\n", 1234567.89);

    return 0;
}
```

Para imprimir caracteres y mensajes (cadenas de caracteres), el formato de la función printf es:

Especificador	Descripción
c	imprime un caracter
s	imprime una cadena de caracteres

```
// Ejemplo formatos_cadenas.c

/**
*  Este programa muestra los diferentes formatos para i
*  imprimir caracteres y cadenas de caracteres
*/

#include <stdio.h>
```

```
int main() {
    char a = 'A';
    char cadena[] = "Esto es una cadena de texto";
    char *cadenap = "Esto tambien es una cadena de texto";

    printf("%c\n", a);
    printf("%s\n", "Esto es una cadena de texto");
    printf("%s\n", cadena);
    printf("%s\n", cadenap);

    return 0;
}
```

3.1.2. funcion puts

Esta función se utiliza para imprimir una cadena de texto. El formato de la función es puts(char *), lo cual significa que recibe un apuntador a una cadena de texto y el ejemplo del programa Hola Mundo queda como:

```
#include <stdio.h>

int main ()
{
  char string [] = "Hello world!";
  puts (string);
}
```

3.2. Entrada por teclado

3.2.1. función scanf

Esta función se puede utilizar para la introducción de cualquier combinación de valores numéricos o caracteres. En términos generales la función scanf se escribe:

```
scanf(cadena de control, arg1,arg2,...,argn);
```

Donde cadena de control hace referencia a una cadena de caracteres que contiene cierta información sobre el formato de los datos y arg1,arg2,...,argn son argumentos que represen-

tan los datos. (En realidad los argumentos representan punteros que indican las direcciones de memoria en donde se encuentran los datos. Esto se verá más adelante).

En la cadena de control se incluyen grupos de caracteres, uno por cada dato de entrada. Cada grupo debe comenzar con el signo de porcentaje, que irá seguido, en su forma más sencilla, de un carácter de conversión que indica el tipo de dato correspondiente.

Carácter de conversión	Significado
c	El dato es un carácter.
d	El dato es un entero decimal.
e	El dato es un valor en coma flotante.
f	El dato es un valor en coma flotante.
g	El dato es un valor en coma flotante.
h	El dato es un entero corto.
i	El dato es un entero decimal, octal o hexadecimal.
o	El dato es un entero octal.
s	El dato es una cadena de caracteres.
u	El dato es un entero decimal sin signo.
x	El dato es un entero hexadecimal.
[...]	El dato es una cadena de caracteres que puede incluir caracteres de espaciado.

Dado que la función scanf requiere de un apuntador a la variable, cada nombre de variable debe ir precedido por un ampersand &. Los datos que se introducen deben corresponderse en tipo y orden con los argumentos de la función scanf. En el caso de que la variable sea un apuntador y/o un arreglo no será necesario colocarlo tal como se muestra en el siguiente ejemplo.

Ejemplo de utilización de scanf:

```
// Ejemplo entrada_scanf.c

/**
* Ejemplo para mostrar el uso de la funcion scanf
*/

#include <stdio.h>

int main () {
  char str [80];
  int i;
```

```
  printf ("Ingrese su nombre : ");
  scanf ("%s",str);
  printf ("Ingrese su edad: ");
  scanf ("%d",&i);
  printf ("Sr. %s , %d aÒos de edad.\n",str,i);
  printf ("Ingrese un numero hexadecimal : ");
  scanf ("%x",&i);
  printf ("Usted dio el numero %#x (%d).\n",i,i);

  return 0;
}
```

3.2.2. función getch

Si lo que queremos es que el usuario introduzca un carácter por el teclado usamos las funciones getch y getche. Estas esperan a que el usuario introduzca un carácter por el teclado. La diferencia entre getche y getch es que la primera saca por pantalla la tecla que hemos pulsado y la segunda no (la e del final se refiere a echo=eco).

Ejemplo:

```
// Ejemplo entrada_caracter.cpp

/**
* Pide un caracter desde el teclado.
*/

#include <stdio.h> #include <conio.h>

int main()

{

  char letra;

  printf( "Introduce una letra: " );


  letra = getche();

  printf( "\nHas introducido la letra: %c", letra );
```

```
  return 0;

}
```

3.2.3. Instrución gets

Esta función es la alternativa para captura de strings que tengan espacios en blanco intermedios cosa que scanf %s no puede hacer. Su formato completo es :

```
gets(variable string);
```

```
// Ejemplo gets_puts.c

/**
*  Ejemplo que muestra el uso de gets y puts
*/

#include <stdio.h>    // Para gets() y puts() #include <conio.h>
// Para clrscr() y gotoxy()


void main() {
  char nombre[31];   // Declara un arreglo de 31 caracteres
  char saludo1[] = "HOLA,"; //Constante de caracteres


  puts("Cual es tu nombre ? "); //Despliega cadena de car.
  gets(nombre);   // Lee cadena de caracteres
  puts(saludo1);
  puts(nombre);

}
```

Un ejemplo donde se muestra la entrada y salida utilizando todos los tipos es el que se muestra a continuación.

```
/*
 * File:   lectura_tipos.c
 * Author: felix
 *
 * Created on 11 de septiembre de 2011, 20:18
```

```
 */

#include <stdio.h>
#include <stdlib.h>

int main() {

    // Para leer un entero

    int d;
    printf("Leyendo un entero d = ");
    scanf("%d", &d);
    printf("El entero leido es %d \n", d);

    // Para leer un caracter

    char c;
    printf("Leyendo un caracter c = ");
    scanf("%s", &c);
    printf("El caracter leido es %c\n", c);

    // Para leer un flotante

    float f;
    printf("Leyendo un Florante f = ");
    scanf("%f", &f);
    printf("El flotante leido es %f\n", f);


    // Para leer una cadena de caracteres

    char s[20];
    printf("Leyendo una cadena de caracteres s = ");
    scanf("%s", s);
    printf("La cadena es %s\n",s);

    return (0);
}
```

3.3. Ejemplos

Utilizando la entrada salida por consola, reescribir los programas de la sección 2.6, para que permitan utilizar valores dados desde teclado.

3.3.1. Cálculo de Áreas

En C

```
// Ejemplo Areas_volumenes.c

/**
 * <p>Title: Calculo de Areas de figuras</p>
 * <p>Description: Calcula el area y volumen de simples figuras utilizando
 * lectura de datos desde el teclado</p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: UMSNH</p>
 * @author Dr. Felix Calderon Solorio
 * @version 1.0
 */

#include <stdio.h>

int main()
{
  // datos
  float base, altura, radio;

  printf("Dame el valor de la base ");
  scanf("%f", &base);

  printf("Dame el valor de altura ");
  scanf("%f", &altura);

  printf("%f\n", altura);

  printf("Dame el radio ");
  scanf("%f", &radio );
```

```
  // areas

  float Atriangulo  = (base * altura) / 2.0;
  float Acuadrado   = base*base;
  float Arectangulo = base * altura;

  // volumenes

  float Abase       = 3.1416*radio*radio;
  float Vcilindro   = Abase * altura ;
  float Vcono       = Abase * altura / 3.0;

  printf("Area de triangulo  = %f\n", Atriangulo);
  printf("Area de cuadrado   = %f\n", Acuadrado);
  printf("Area de rectangulo = %f\n", Arectangulo);
  printf("Volumen cilindro   = %f\n", Vcilindro);
  printf("Volumen cono       = %f\n", Vcono);

  return 0;
}
```

3.3.2. Ley de Ohm y Análisis de transistores

Resolver el problema de la sección 2.6 utilizando instrucciones de entrada salida.

En C

```
// Ejemplo ec037.cpp

/**
 * Title: Calculo de la ganancia de voltaje para un transistor
 * Description: Programa para determinar ganancias de voltaje utilizando dato
 * desde el teclado
 * Copyright: Copyright (c) 2005
 * Company: UMSNH
 * author Dr. Felix Calderon Solorio
 * version 1.0
 */
```

```
#include <stdio.h>

int main() {

  float Vi, Re, Rc, R;
  float I, Il, Vl, Av;

  printf("Dame el voltaje de entrada Vi = ");
  scanf("%f", &Vi);
  printf("Dame el valor de la Resistencia de emisor Re = ");
  scanf("%f", &Re);
  printf("Dame el valor de la Resistencia de colector Rc = ");
  scanf("%f", &Rc);
  printf("Dame el valore de la Resistencia de carga Rl = ");
  scanf("%f", &R);

  // calculo de la corriente de emisor.

  I = Vi/Re;
  printf("I = %f\n", I);

  // Asumiendo una ganacia alpha = 1 y que Ic = Ie

  Il = I;
  printf("Il = %f\n", Il);

  // el voltaje en la resistencia R es

  Vl = Il*R;
  printf("Vl = %f\n", Vl);


  // finalmente la ganacia de voltaje es
   Av = Vl/Vi;
  printf("Av = %f\n", Av);

  return 0;
}
```

3.3.3. Ejemplo Mayor de edad

El siguiente código permite determinar si una persona es mayor de edad en el año 2011 dependiendo del año en que nació.

```
/*
 * File:   mayor_edad.c
 * Author: felix
 *
 * Created on 11 de septiembre de 2011, 20:36
 */

#include <stdio.h>
#include <stdlib.h>

int main() {
    int edad, anio;
    printf("Dame el anio en que naciste ");
    scanf("%d", &anio);
    edad = 2011-anio;
    char mayor = 'S'*(edad>=18) + 'N'*(edad<18);
    printf("\n%c\n", mayor);

    return (0);
}
```

Sentencias de condición y de repetición

4.1. Sentencias de Condición

4.1.1. if-else

En cualquier lenguaje de programación es habitual tener que comprobar si se cumple una cierta condición. La forma normal de conseguirlo es empleando una construcción de la forma:

```
SI   condición_se_cumple   ENTONCES   pasos_a_dar
SINO pasos_alternos
```

Esta construcción se encuentra en la mayoría de los lenguajes y es conocida como intrusiones condicionales. En el caso de C, la forma exacta será empleando if (si, en inglés), seguido por la condición entre paréntesis, e indicando finalmente entre llaves los pasos que queremos dar si se cumple la condición, así :

if (condición) sentencias

Por ejemplo,

```
if (x == 3)  {
  printf( "El valor es correcto\n" );
  resultado = 5;
}
```

Nota: Si solo queremos dar un paso en caso de que se cumpla la condición, no es necesario emplear llaves. Las llaves serán imprescindibles solo cuando haya que hacer varias cosas:

```
if (x == 3)
  printf( "El valor es correcto\n" );
```

Una primera mejora, que también permiten muchos lenguajes de programación, es indicar qué hacer en caso de que no se cumpla la condición. Sería algo parecido a

```
SI  condición_a_comprobar  ENTONCES  pasos_a_dar
EN_CASO_CONTRARIO  pasos_alternativos
```

que en C escribiríamos así:

```
if (condición)  { sentencias1 }   else   { sentencias2 }
```

Por ejemplo,

```
if (x == 3)  {
  printf( "El valor es correcto\n" );
  resultado = 5;
}
else  {
  printf( "El valor es incorrecto\n" );
  resultado = 27;
}
```

Ejemplo

El siguiente ejemplo permite calcular dado tres números cual es el mayor y cual es el menor de ellos.

```
// Ejemplo mayor_menor.cpp

/**
 * Title: Mayor y menor
 * Description: Dados tres numeros dice cual es el mayor y cual el menor
 * Copyright: Copyright (c) 2003
 * Company: UMSNH
 * author Dr. Felix Calderon Solorio
 * version 1.0
 */


#include <stdio.h>

int main ()
{
```

```
  int a, b, c, d;

  a = 4; b = 2; c = 3;
  d = (a > b) ?  a : b;
  d = (d > c) ?  d : c;

  printf("el Mayor es %d\n", d);


  // otra forma

  if(a> b) d = a;
  else d = b;
  if(d< c) d = c;

  printf("el Mayor es %d\n", d);

  return 0;
}
```

Ejemplo

Este ejemplo nos permite determinar dado los tres lados de un triángulo, si es un triángulo Equilátero, Escaleno Isósceles

```
/*
 * File:   triangulos_1.c
 * Author: felix
 *
 * Created on 15 de septiembre de 2011, 11:25
 */

#include <stdio.h>
#include <stdlib.h>

/*
 * Determinar el tipo de triangulo a partir de los lados a b y c
 */
int main(int argc, char** argv) {
    float a,b,c;
```

```
    printf("Por favor dame el primer lado a =  ");
    scanf("%f", &a);
    printf("Por favor dame el segundo lado b = ");
    scanf("%f", &b);
    printf("Por favor dame el tercer lado c =  ");
    scanf("%f", &c);

    printf("\n");

    // Comienza calculando la desigualdad del triangulo ya que
    // a + b > = c para todos los lados

    if((a+b > c) && (a+c >b) && (b+c >a)) {
        // en el caso de un triangulo equilatero
        if(a==b && a==c) printf("Es un triangulo Equilatero\n");
        // en el caso de un triangylo Isoseles
        else if (a==b || b==c || a==c) printf("Es un triangulo Isoseles\n");
        // en el caso de un triangulo escaleno
        else printf("Es un triangulo Escaleno\n");
    }
    else printf("Para los valores que me diste no forman un triangulo\n");

    return (0);
}
```

Ejemplo

Otro ejemplo basado en triángulos es el siguiente. En este se calcula además del tipo de triángulo los ángulos internos del mismo

```
// Ejemplo triangulos.c

/**
 * Title: Tipos de triangulos
 * Description: Programa para determinar el tipo de triangulo dados tres lados
 * Copyright: Copyright (c) 2003
 * Company: UMSNH
 * author Dr. Felix Calderon Solorio
 * version 1.0
 */
```

```
#include <stdio.h>
#include <math.h>

int main()
{
  double a = 1, b = 1, c = 1; // tamaño de los lados
  double A, B, C; // angulos
  double suma;
  double PI = 3.14159265358979323846;

  A = acos( (b * b + c * c - a * a) / (2 * b * c)) * 180 / PI;
  B = acos( (a * a + c * c - b * b) / (2 * a * c)) * 180 / PI;
  C = acos( (a * a + b * b - c * c) / (2 * a * b)) * 180 / PI;

  suma = A + B + C;

  printf("Los lados son   = [ %f %f %f ]\n", a, b, c);
  printf("Los angulos son = [ %f %f %f ]\n", A, B, C);
  printf("La suma de sus angulos es = %f\n", suma);


  if (abs(suma - 180) < 1e-6) {
    if (a == b && b == c)
      printf("Es un triangulo equilatero \n");
    else {
      if (A == 90 || B == 90 || C == 90)
        printf("Es un triangulo rectangulo \n");
      else {
        if (a == b || b == c || a == c)
          printf("Es un triangulo isoceles \n");
        else {
          if (A != B && B != C && C != A)
            printf("Es un triangulo escaleno \n");
        }
      }
    }
  }
  else
    printf("Estos valores no forman un triangulo \n");
```

```
  return 0;
}
```

Ejemplo

Para una ecuación cuadrática de la forma $f(x) = Ax^2 + Bx + C$, calcular los de x que hacen $f(x) = 0$ utilizando la formula general

$$x = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

La solución en C queda

```
// Ejemplo eq_cuadratica.cpp

/**
 * Title: Raices de un Polinomio
 * Description: este ejemplo calcula las raices de una ecuacion de segundo grado
 * Copyright: Copyright (c) 2003
 * Company: UMSNH
 * author Dr. Felix Calderon Solorio
 * version 1.0
 */

#include <stdio.h>
#include <math.h>

int main()
{

  double A=9, B=6, C=4;
  double X1=0, X2=0, G=0, I=0, D=0;

  D=(B*B-(4*A*C));

  if (D>0)
  {
    // si las raices son positivas

    X1=(-B+sqrt(D))/(2*A);
    X2=(-B-sqrt(D))/(2*A);
```

```
    printf("X1 = %f\n", X1);
    printf("X2 = %f\n", X2);
  }
  else
  {
    // si las raices son negativas
    G=-B/(2*A);
    I=sqrt(-D)/(2*A);
    printf("X1 = %f, %f j\n", G, I);
    printf("X2 = %f, %f j\n", G, -I);
  }

  return 0;
}
```

Ejemplo

Un objeto es lanzado con una velocidad inicial $v_i = 20$ m/s hacia arriba. Determinar a) La altura máxima a la que llega y b) los tiempos en que el objeto esta a la mitad de la altura máxima.

```
/*
 * File:   caida_libre.c
 * Author: felix
 *
 * Created on 15 de septiembre de 2011, 10:47
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char** argv) {
    float vi=20, g = 9.8, hmax;

    float A, B, C, D, t1, t2, aux;

    // a) Utilizando h = (vi^2 - vf^2)/(2*g)  la hmax ocurre cuando vf = 0

    hmax = vi*vi/2.0/g;
    printf("La velocidad inicial es %f\n", vi);
```

```
    printf("La altura Maxima es %f\n\n", hmax);

    // b) Para h = hmax/2 Resolvemos
    // y = vi*t - 0.5*g*t^2 con h = hmax/2
    // 0.5*g t^2 - vi t + hmax/2 = 0

    A = 0.5*g;
    B = -vi;
    C= hmax/2.0;

    D = B*B - 4.0*A*C;

    if(D < 0) printf("No existe solucion al problema\n") ;
    else {
        t1 = (-B + sqrt(D))/2.0/A;  // t = (- B +- (B^2 - 4AC)^0.5)/(2.0*A)
        t2 = (-B - sqrt(D))/2.0/A;

        if(t1>t2) {
            aux = t1;
            t1 = t2;
            t2 = aux;
        }

        printf("El objeto esta a la altura h = %f  \n\n", C);
        printf("Cuando sube en t1 = %f y cuando baja en t2 = %f\n", t1, t2);
    }

    return (0);
}
```

Ejemplo

Determinar si la suma de un número de 4 digitos es igual a 21

```
/*
 * File:   suma_21.c
 * Author: felix
 *
 * Created on 20 de septiembre de 2011, 18:44
 */
```

```
#include <stdio.h>
#include <stdlib.h>

/*
 * Escribir un programa que permita calcular si las ultimas 4 cifras de un
 * numero suma 21
 */

int main(int argc, char** argv) {
    int n1, n2, n3, n4, suma, n, n0;
    printf("Dame un numero entero ");
    scanf("%d", &n0);

    n = n0%10000;    // se queda solamente con la ultimas 4 cifras de n

    n1 = n/1000;    // calcula n1 como la division entera
    n %= 1000;      // se queda solamente con las ultimas 3 cifras de n

    n2 = n/100;     // calcula n2 como la division entera
    n %= 100;       // se queda solamente con las ultimas 2 cifras de n

    n3 = n/10;      // calcula n3 como la division entera
    n4 = n%10;      // se queda con el utimo digito

    suma = n1+n2+n3+n4;
    if(suma == 21) printf("La suma es igual a 21\n");
    else printf("Sigue intentando la suma es %d\n", suma);

    return (0);
}
```

4.1.2. switch

Si queremos comprobar varias condiciones, podemos utilizar varios if - else - if - else - if encadenados, pero tenemos una forma más elegante en C de elegir entre distintos valores posibles que tome una cierta expresión. Su formato es :

```
  switch (expresion) {
    case valor1: sentencias1; break;
    case valor2: sentencias2; break;
```

```
    case valor3: sentencias3; break;
    ...                                  // Puede haber más valores
}
```

Ejemplo

Como ejemplo podemos ver la siguiente implementación en C, que permite calcular cual es el último dígito de un número dado

```
/*
 * File:   switch.c
 * Author: felix
 *
 * Created on 13 de septiembre de 2011, 19:57
 */

#include <stdio.h>
#include <stdlib.h>

/*
 * Este programa calcula las unidades en un numero entero
 */
int main(int argc, char** argv) {
    int d = 243;

    switch(d%10) {
        case 0: printf("cero ");   break;
        case 1: printf("uno ");    break;
        case 2: printf("dos ");    break;
        case 3: printf("tres ");   break;
        case 4: printf("cuatro "); break;
        case 5: printf("cinco ");  break;
        case 6: printf("seis ");   break;
        case 7: printf("siete ");  break;
        case 8: printf("ocho ");   break;
        case 9: printf("nueve ");  break;
        default : ;
    }

    printf("Unidades\n");
```

```
    return (0);
}
```

Es decir, después de la orden switch indicamos entre paréntesis la expresión que queremos evaluar. Después, tenemos distintos apartados, uno para cada valor que queramos comprobar; cada uno de estos apartados se precede con la palabra case, indica los pasos a dar si es ese valor el que se da, y terminar con break.

Un ejemplo sería:

```
  switch ( x * 10) {
    case 30: printf( "El valor de x era 3" ); break;
    case 50: printf( "El valor de x era 5" ); break;
    case 60: printf( "El valor de x era 6" ); break;
}
```

también podemos indicar qué queremos que ocurra en el caso de que el valor de la expresión no sea ninguno de los que hemos detallado:

```
  switch (expresion) {
    case valor1: sentencias1; break;
    case valor2: sentencias2; break;
    case valor3: sentencias3; break;
    ...                                  // Puede haber más valores
    default: sentencias;                 // Opcional: valor por defecto
}
```

Por ejemplo, así:

```
  switch ( x * 10) {
    case 30: printf( "El valor de x era 3" ); break;
    case 50: printf( "El valor de x era 5" ); break;
    case 60: printf( "El valor de x era 6" ); break;
    default: printf( "El valor de x no era 3, 5 ni 6" ); break;
}
```

Podemos conseguir que en varios casos se den los mismos pasos, simplemente eliminado la orden “break” de algunos de ellos. Así, un ejemplo algo más completo podría ser:

```
  switch ( x ) {
    case 1:
    case 2:
    case 3:
```

```
        printf( "El valor de x estaba entre 1 y 3" );
        break;
    case 4:
    case 5:
        printf( "El valor de x era 4 o 5" );
        break;
    case 6:
        printf( "El valor de x era 6" );
        valorTemporal = 10;
        printf( "Operaciones auxiliares completadas" );
        break;
    default:
        printf( "El valor de x no estaba entre 1 y 6" );
        break;
}
```

Ejemplo

El siguiente ejemplo permite calcular, dada una letra en que rango se encuentra.

```
/*
 * File:   rango_letras.c
 * Author: felix
 *
 * Created on 13 de septiembre de 2011, 20:06
 */

#include <stdio.h>
#include <stdlib.h>

/*
 * Uso de la sentencia switch. El programa clasifica una letra por rangos
 */

int main(int argc, char** argv) {

    char c;
    printf("Dame un caracter ");

    c = getchar();
```

```
    if(c >= 'A' && c <= 'Z') c += 32;

    switch(c) {
        case 'a' :
        case 'b' :
        case 'c' :
        case 'd' : printf("La letra esta entre A y D\n"); break;
        case 'e' :
        case 'f' :
        case 'g' :
        case 'h' : printf("La letra esta entre E y H\n"); break;
        case 'i' :
        case 'j' :
        case 'k' :
        case 'l' : printf("La letra esra ebtre I y L\n"); break;
        case 'm' :
        case 'n' :
        case 'o' :
        case 'p' : printf("La letra esra ebtre M y P\n"); break;
        case 'q' :
        case 'r' :
        case 's' :
        case 't' : printf("La letra esra ebtre Q y T\n"); break;
        case 'u' :
        case 'v' :
        case 'w' :
        case 'x' : printf("La letra esra ebtre U y X\n"); break;
        case 'y' :
        case 'z' : printf("La letra esra ebtre X y Z\n"); break;
        default : printf("No es un caracter valido\n");
    }

    return (0);
}
```

Un poco más adelante propondremos ejercicios que permitan afianzar todos estos conceptos.

4.1.3. Operador Condicional

Existe una construcción adicional, que permite comprobar si se cumple una condición y devolver un cierto valor según si dicha condición se cumple o no. Es lo que se conoce como el "operador condicional (?)":

```
condicion ? resultado_si cierto : resultado_si_falso
```

Es decir, se indica la condición seguida por una interrogación, después el valor que hay que devolver si se cumple la condición, a continuación un símbolo de "dos puntos" y finalmente el resultado que hay que devolver si no se cumple la condición.

Es frecuente emplearlo en asignaciones (aunque algunos autores desaconsejan su uso porque puede resultar menos legible que un "if"), como en este ejemplo:

x = (a == 10) ? b*2 : a ;

En este caso, si a vale 10, la variable tomará el valor de b*2, y en caso contrario tomará el valor de a. Esto también se podría haber escrito de la siguiente forma, más larga pero más legible:

```
if (a == 10)  x = b*2; else x = a;
```

El siguiente es un ejemplo donde se utilizan las expresiones condicionales.

```
/*
 * File:   expresion_condicional.c
 * Author: felix
 *
 * Created on 13 de septiembre de 2011, 13:06
 */

#include <stdio.h>
#include <stdlib.h>

/*
 * Este es un ejemplo de como utilizar expresiones condicionales.
 */
int main() {

    int x =1, y;

    /*
     * La condicional
    */
```

```
    if(x==1) y = 20;
    else y = 10;

    printf("y = %d\n", y);

    /*
     * se puede escribir como
     */

    y=0;

    y = (x==1) ? 20 : 10;
    printf("y = %d\n", y);

    /*
     *Otro ejemplo es
     */

    if(x==1) puts("en Auto");
    else puts("en bici");

    /**
     *Utilizando expresiones
     */

    puts((x==1) ? "en Auto" : "en bici");

    return (0);
}
```

4.2. Repetición

Con frecuencia tendremos que hacer que una parte de nuestro programa se repita (algo a lo que con frecuencia llamaremos bucle), bien mientras se cumpla una condición o bien un cierto número prefijado de veces.

4.2.1. while

C incorpora varias formas de conseguirlo. La primera que veremos es la orden while, que hace que una parte del programa se repita mientras se cumpla una cierta condición. Su formato será:

```
while (condición) {
    sentencia 1;
    sentencia 2;
    ...
 }
```

Es decir, la sintaxis es similar a la de if, con la diferencia de que aquella orden realizaba la sentencia indicada una vez como máximo (si se cumple la condición), pero while puede repetir la sentencia más de una vez (mientras la condición sea cierta). Al igual que ocurría con if, podemos realizar varias sentencias seguidas (dar más de un paso) si las encerramos entre llaves:

```
x = 20;
while ( x > 10) {
  printf("Aun no se ha alcanzado el valor limite\n");
  x --;
}
```

4.2.2. do-while

Existe una variante de este tipo de bucle. Es el conjunto do..while, cuyo formato es:

```
do {
  sentencia1;
  sentencia2;
  ...
} while (condición)
```

En este caso, la condición se comprueba al final, lo que quiere decir que las "sentencias" intermedias se realizarán al menos una vez, cosa que no ocurría en la construcción anterior (un único while antes de las sentencias), porque si la condición era falsa desde un principio, los pasos que se indicaban a continuación de while no llegaban a darse ni una sola vez.

Un ejemplo típico de esta construcción "do..while" es cuando queremos que el usuario introduzca una contraseña que le permitirá acceder a una cierta información:

```
do {
```

```
    printf("Introduzca su clave de acceso\n");
    scanf("%f", &claveIntentada);  // LeerDatosUsuario realmente no existe
} while (claveIntentada != claveCorrecta)
```

En este ejemplo hemos la función scanf para leer un número flotante.

4.2.3. for

Una tercera forma de conseguir que parte de nuestro programa se repita es la orden for. La emplearemos sobre todo para conseguir un número concreto de repeticiones. Su formato es

```
for ( valor_inicial ; condicion_continuacion ; incremento ) {
  sentencias1;
  sentencias2;
  ...
}
```

(es decir, indicamos entre paréntesis, y separadas por puntos y coma, tres Ordenes: la primera dará el valor inicial a una variable que sirva de control; la segunda orden será la condición que se debe cumplir para que se repitan las sentencias; la tercera orden será la que se encargue de aumentar -o disminuir- el valor de la variable, para que cada vez quede un paso menos por dar).

Esto se verá mejor con un ejemplo. podríamos repetir 10 veces un bloque de Ordenes haciendo:

```
for ( i=1 ; i<=10 ; i++ ) {
  ...
}
```

O bien podríamos contar descendiendo desde el 20 hasta el 2, con saltos de 2 unidades en 2 unidades, así:

```
for ( j = 20 ; j > 0 ; j -= 2 )
  printf( "%d\n", j );
```

Nota: se puede observar una equivalencia casi inmediata entre la orden for y la orden while. Así, el ejemplo anterior se podría reescribir empleando while, de esta manera:

```
j = 20;
while ( j > 0 )
  printf( "%d\n", j );
  j -= 2;
```

```
}
```

Precauciones con los bucles: Casi siempre, nos interesará que una parte de nuestro programa se repita varias veces (o muchas veces), pero no indefinidamente. Si planteamos mal la condición de salida, nuestro programa se puede quedar colgado, repitiendo sin fin los mismos pasos.

Se puede modificar un poco el comportamiento de estos bucles con las ordenes break y continue.

La sentencia break hace que se salten las instrucciones del bucle que quedan por realizar, y se salga del bucle inmediatamente. Como ejemplo:

```
for ( i=1 ; i<=10 ; i++ ) {
  printf( "Comenzada la vuelta " );
  printf( "%d\n", i );
  if (i==8)  break;
  printf( "Terminada esta vuelta\n" );
}
printf( "Terminado\n"  );
```

En este caso, no se mostraría el texto Terminada esta vuelta para la pasada con i=8, ni se darían las pasadas de i=9 e i=10, porque ya se ha abandonado el bucle.

La sentencia continue hace que se salten las instrucciones del bucle que quedan por realizar, pero no se sale del bucle sino que se pasa a la siguiente iteración (la siguiente vuelta o pasada). Como ejemplo:

```
for ( i=1 ; i<=10 ; i++ ) {
  printf( "Comenzada la vuelta\n" );
  printf( "%d\n", i );
  if (i==8)  continue;
  printf( "Terminada esta vuelta\n" );
}
printf( "Terminado\n"  );
```

En este caso, no se mostraría el texto Terminada esta vuelta para la pasada con i=8, pero si se darían la pasada de i=9 y la de i=10.

también esta la posibilidad de usar una etiqueta para indicar donde se quiere saltar con break o continue. Solo se debería utilizar cuando tengamos un bucle que a su vez está dentro de otro bucle, y queramos salir de golpe de ambos. Es un caso poco frecuente, así que no profundizaremos más, pero si veremos un ejemplo:

```
for ( i=1 ; i<=10 ; i++ ) {
  printf( "Comenzada la vuelta\n" );
```

```
      printf( "%d\n", i );
      if (i==8)  break salida;
      printf( "Terminada esta vuelta\n" );
    }
    printf( "Terminado\n"  );
    salida:
    ...
```

En este caso, a mitad de la pasada 8 se saltaría hasta la posición que hemos etiquetado como salida (se define como se ve en el ejemplo, terminada con el símbolo de dos puntos), de modo que no se escribiría en pantalla el texto Terminado (lo hemos saltado).

Ejemplo

Hacer un programa que permita calcular la siguiente sumatoria

$$S = \sum_{i=0}^{i=64} 2^i$$

La implementación en C queda:

```
// Ejemplo sumatoria.c

/**
 * Title: Sumatoria
 * Description: Este programa calcula la suma de 1 + 2 + 4 + 8+ 16 + ...2n
 * Copyright: Copyright (c) 2003
 * Company: UMSNH
 * author Dr. Felix Calderon Solorio
 * version 1.0
 */

#include <stdio.h>


int main ()
{
  int i=1, s=0;

  // utilizando for()
```

```
  for(i=0; i<=30; i++){
    s=((2*s) +1);
    printf("%d s=%d\n", i, s);
  }

  // utilizando while

  i = 1;
  s = 0;
  while(i <=30)
  {
    s=((2*s) +1);
    printf("%d s=%d\n", i, s);
    i++;
  }

  // utilizando do-while

  i = 1;
  s = 0;
  do
  {
    s=((2*s) +1);
    printf("%d s=%d\n", i, s);
    i++;
  }
  while (i<=30);
  return 0;
}
```

Ejemplo

El siguiente ejemplo permite hacer una comparación entre la implementación utilizando for y do-while para leer una clave desde teclado.

```
/*
 * File:   clave.c
 * Created on 22 de septiembre de 2011, 12:24
 */
```

```
#include <stdio.h>
#include <stdlib.h>

/*
 * Simula el proceso de acceso a mediante clave a un cajero
 */
int main(int argc, char** argv) {
    int clave;

    // impelementacion utilizando for

    for(;;){
        printf("Dame tu clave ");
        scanf("%d", &clave);
        if(clave == 1234) break;
    }

    // implementacion con do-while

    do {
        printf("Remita su clave nuevamente ");
        scanf("%d", &clave);
    } while (clave != 1234);

    printf("Bienvenido\n");

    return (0);
}
```

Ejemplo

Escribir un programa que permita imprimir todas las combinaciones de palabra de tres letras que se pueden escribir utilizando las letras del alfabeto. Así por ejemplo la salida de este código es

AAA AAB AAC ... ABA ABC

etc.

```
/*
 * File:   combinaciones.c
 * Author: felix
```

```
 *
 * Created on 5 de octubre de 2011, 11:27
 */

#include <stdio.h>
#include <stdlib.h>

/*
 *
 */
int main(int argc, char** argv) {
    int i, j, k, n=1;
    int letra = 'A';

    for(i=0; i<3; i++){
        for(j=0; j<3; j++){
            for(k=0; k<3; k++)
                printf("%02d.- %c%c%c \n", n++, letra +i, letra+j, letra +k);
        }
    }

    return (EXIT_SUCCESS);
}
```

Ejemplo

Escribir un programa que permita hacer la siguiente tabulación correspondiente a la función $f(x) = x^2$

x	$f(x)$
1	1
2	4
3	9
4	16

```
//Ejemplo funcion.c

/**
 * <p>Title: Ejemplo de una funcion</p>
 * <p>Description: Realiza la funcion x2</p>
```

```
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: UMSNH</p>
 * @author Dr. Felix Calderon Solorio
 * @version 1.0
 */

#include <stdio.h>

float cuadrado(float); // prototipo de funcion

int main()
{
  int x;

  for(x=1; x<=10; x++)
    printf("%d^2 = %f \n", x, cuadrado(x));

  return 0;
}

// Definicion  de la funcion

float cuadrado(float y)
{
  return y*y;
}
```

Ejemplo

Escribir un programa que permita imprimir un triángulo como el que se muestra a continuación

```
*
**
***
****
*****
******
*******
```

```
/*
 * File:   triangulo_asteriscos.c
 * Author: felix
 *
 * Created on 5 de octubre de 2011, 11:22
 */

#include <stdio.h>
#include <stdlib.h>

/*
 *
 */
int main(int argc, char** argv) {
    int i, j;

    for(i=0; i<20; i++) {
        for(j=0; j<i; j++)
            printf("*");
        printf("\n");
    }

    return (0);
}
```

4.3. Ejemplo de sucesiones e Iteración de punto fijo

4.3.1. Métodos iterativos para resolver $x = g(x)$

Necesitamos una regla, fórmula o función $g(x)$ con la cual calcularemos temimos sucesivos, para un valor de partida P_0. Lo que se produce es una sucesión de valores P_k obtenida mediante el proceso iterativo $P_{k+1} = g(P_k)$. La sucesión se ajusta al siguiente patrón:

$$P_1 = g(P_0)$$

$$P_2 = g(P_1)$$

$$P_3 = g(P_2)$$

$$\dots$$

$$P_{k+1} = g(P_k)$$

Lo que podemos observar de ésta sucesión es que si $P_{k+1} \leftarrow P_k$ la sucesión converge a un valor, pero en el caso de que P_{k+1} no tienda a P_k tenemos una sucesión divergente.

Ejemplo

Dada la sucesión $P_{k+1} = A_k$ con $P_0 = 1$, producirá:

$$P_1 = A$$
$$P_2 = A * A = A^2$$
$$P_3 = A * A^2 = A^3$$
$$...$$
$$P_k = A^k$$

4.3.2. Métodos de punto fijo

Un punto fijo de una función g(x) es un número real P tal que $P = g(P)$. Geométricamente hablando, los puntos fijos de una función $g(x)$ son los puntos de intersección de la curva $y = g(x)$ con la recta $y = x$.

Ejemplo

Consideremos la función $p = e^{-p}$ y damos un valor inicial $p_0 = 0{,}5$

La solución de esta recurrencia es:

$$p_1 = e^{-0,5} = 0{,}606531 p_2 = 0{,}545339 p_3 = 0{,}579703 ... p_k = 0{,}567143$$

La implementación en C es

```
/*
 * File:   punto_fijo.c
 * Author: felix
 *
 * Created on 2 de marzo de 2012, 12:35
 */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char** argv) {
    int iter = 0;
    double x0, x1;

    x0 = x1 = 0.5;

    do
    {
        printf("X(%d)= %f\n", iter, x1);
        x0 = x1;
        x1 = exp(-x0);
        iter ++;
    } while (fabs((x0-x1)/x1) > 0.001);

    return (EXIT_SUCCESS);
}
```

4.3.3. Solución de Sistemas no lineales

Consideremos el caso de la funciones :

$$f_1(x, y) = x^2 - 2x - y + 0{,}5$$

$$f_2(x, y) = x^2 + 4y^2 - 4$$

y queremos encontrar un método para calcular los ceros de estas funciones, es decir, los valores x y y que hacen $f_1(x, y) = 0$ y $f_2(x, y) = 0$

Podemos resolver el sistema, utilizando el método de iteración de punto fijo. Para ello despejamos de la ecuación 1 a x:

$$x = \frac{x^2 - y + 0{,}5}{2}$$

A la segunda ecuación agregamos un termino

$$x^2 + 4y^2 - 8y - 4 = -8y$$
$$y = \frac{-x^2 - 4y^2 + 8y + 4}{8}$$

De esto tenemos un sistema en sucesión dado por:

$$x_{k+1} = \frac{x_k^2 - y_k + 0{,}5}{2}$$
$$y_{k+1} = \frac{-x_k^2 - 4y_k^2 + 8y_k + 4}{8}$$

La solución de esta sucesión para un valor inicial $[0, 1]$ es:

k	x	x
0	0.0000	1.0000
1	-0.2500	1.0000
2	-0.2188	0.9922
3	-0.2222	0.9940
4	-0.2223	0.9938
5	-0.2222	0.9938
6	-0.2222	0.9938

4.4. Salto

El uso del GOTO implica un salto incondicional de un lugar a otro del programa. Esta práctica hace que los programas sean muy difíciles de corregir ó mantener. Si no quedara más remedio que usarlo, (y en programación estructurada SIEMPRE hay remedio) debe marcarse el destino del salto mediante un nombre seguido por dos puntos.

```
if( c == 0 )  goto OTRO_LADO;

.............................

.............................

OTRO_LADO: printf(........
```

En este caso si c es cero se saltan todas las sentencias entre el if y el destino, continuándose con la ejecución del printf() . El destino puede ser posterior como anterior al GOTO invocante.

Programación estructurada con funciones

5.1. Introducción.

En C existen un conjunto de funciones matemáticas de biblioteca las que permiten al programador ciertos cálculos comunes. Algunos ejemplos de estas funciones son:

función	descripción
double acos(double x)	arco coseno de x.
double asin(double x)	arco seno de x.
double atan(double x)	arco tangente en radianes.
double atan2(double y, double x)	arco tangente de las dos variables x e y.
double ceil(double x)	Redondea x hacia arriba al entero más cercano.
double cos(double x)	coseno de x.
double cosh(double x)	coseno hiperbólico de x.
double exp(double x)	Devuelve el valor de e
double fabs(double x)	valor absoluto
double floor(double x)	Redondea x hacia abajo al entero más cercano.
double fmod(double x, double y)	Calcula el resto de la divisiÛn de x entre y.
double log(double x)	Devuelve el logaritmo neperiano de x.
double log10(double x)	Devuelve el logaritmo decimal de x.
double pow(double x, double y)	Devuelve el valor de x elevado a y.
double sin(double x)	Devuelve el seno de x.
double sinh(double x)	Regresa el seno hiperbólico de x.
double sqrt(double x)	Devuelve la raÌz cuadrada no negativa de x.
double tan(double x)	Devuelve la tangente de x.
double tanh(double x)	Devuelve la tangente hiperbólica de x.

Las funciones permiten al programador modularizar un programa. Todas las variables declaradas en las definiciones de funciones son variable locales, son conocidas solo en la función en la cual están definidas. La mayor parte de las funciones tienen una lista de parámetros. Los parámetros proporcionan la forma de comunicar información entre fun-

ciones. Los parámetros de una función también son variable locales.

5.2. Definición de una función

El formato para definir una función es

```
tipo_de_valor_de_regreso nombre_de_la_función (lista de parametros)
{

declaraciones;

enunciados;

}
```

El nombre de la función comienza con una letra y continua con letra numero o el carácter "_", nunca debe dejarse espacios en blanco. El tipo de valor de regreso es el tipo de los datos del resultado que una función devolverá por valor. Un tipo de valor de regreso no especificado será siempre supuesto por el compilador cono int (en el caso de C).

La manera de regresar un valor es utilizando la sentencia

```
return expresion;
```

5.2.1. Ejemplo de la función cuadrado

A continuación se presenta la implementación de una simple función que calcula el cuadrado de un numero flotante.

```
/*
 * File:   cuadrado.c
 * Author: felix
 *
 * Created on 28 de marzo de 2012, 13:54
 */

#include <stdio.h>

float cuadrado(float y) {
```

```
  return y*y;
}

int main() {
  int x;

  for(x=1; x<=10; x++)
    printf("%f \n", cuadrado(x));

  return 0;
}
```

5.3. Estructura de una función y alcance de una variable

El objetivo de dividir un programa en funciones es controlar las variables internas usadas. Así por ejemplo el siguiente código presenta la variable x y los valores que esta tendrá son totalmente diferentes.

```
uno() {
   int x = 2;
   ----
   ----
 }

dos() {
   int x = 4;
   -----
   -----
 }
```

En el caso de la función uno(), la variable x se iniciativa con 2 y esta solamente existirá en dicha función. Por otro lado en la función dos() la variable x tendrá un 4 y no interfieren una variable con la otra. Podemos decir que la primer x pertenece a la función uno y la segunda a la función dos.

5.3.1. Prototipos de función

Un prototipo de función le indica al compilador el tipo de dato regresado por la función, el número de parámetros que la función espera recibir, los tipos de dichos parámetros y el orden en que se esperan dichos parámetros.

5.3.2. Ejemplo de la función máximo

Realizar un programa que permita calcular para tres números enteros cual es el mayor y cual es el menor. La implementación en C y Java se muestra a continuación, note que solamente en C, es necesario declarar los prototipos de función.

```
/*
 * File:   maximo_minimo.c
 * Author: felix
 *
 * Created on 28 de marzo de 2012, 13:57
 */

#include <stdio.h>
#include <stdlib.h>

/*
 *
 */

// Prototipo de Funcion

int maximo(int x, int y, int z);

// funcion principal

int main() {
  int a, b, c, max;

  printf("Dar tres numeros enteros ");
  scanf("%d%d%d", &a, &b, &c);

  max = maximo(a,b,c);
```

```
  printf("El maximo es %d\n", max);

  return 0;
}

// definicion de la funcion

int maximo(int x, int y, int z)
{
  int max = x;

  if(y > max) max = y;
  if(z > max) max = z;

  return max;
}
```

5.4. Paso de argumentos a una función y tipo de dato devuelto

La definición matemática de una función es $f : \mathbb{R}^N \to \mathbb{R}$. Esto significa que la función f recibe un vector de números reales de tamaño N (variable independientes) y regresa un número real resultado de la evaluación. En el caso de las funciones en C tenemos que los valores independientes se ponen entre los paréntesis indicando el tipo de variable de cada una y el tipo de la variable que devuelve se pone al comienzo de la función

Así por ejemplo cuando escribimos la función $f(x) = x^2$ en C tenemos que escribir

```
float f(float x)
{
   return x*x;
}
```

Al igual que la definición la función se llama f y recibe un número real x y regresa un número real que el el cuadrado x^2.

Otro ejemplo es calcular la suma de dos variables a y b la función en este caso recibe dos argumentos a y b y regresa el valor de $a + b$. La codificación en C es:

```
int suma (int a, int b)
{
   return (a+b);
}
```

5.4.1. Argumentos de main

En general la declaración de la función main esta dada como

```
int main (int argc, char **argv);
```

Lo cual indica que la función main regresa un entero. En algunos sistemas operativos como Windows o Unix, este valor es regresado por la linea de comando dando instrucciones de que el código se ejecuto sin error, para ello enviamos el código EXIT_SUCCESS o simplemente un cero

```
int main (int argc, char **argv){
   return 0;
 }
```

Existen al menos dos argumentos de main: argc y argv. El primero de estos es un contador de los argumentos enviados al programa y el segundo un arreglo de cadenas de caracteres con estos argumentos. Estos argumentos son pasados al programa por el sistema operativo en la linea de comando al ejecutar el programa. Así por ejemplo cuando generamos un programa al que llamaremos argumentos_main, al ejecutarlo tendremos

```
./argumentos_main Hola Mundo Cruel
```

Las variable argc tendrá un 4 como valor y argv será un arreglo de caracteres con las palabras argumentos_main, Hola, Mundo y Cruel. El siguiente código es un ejemplo de como visualizar los argumentos de main.

```
/*
 * File:   argumentos_main.c
 */

#include <stdio.h>
#include <stdlib.h>

/*
 *
 */
```

```
int main(int argc, char** argv) {
    int i;
    printf("El numero de argumentos es %d\n", argc);

    for(i=0; i<argc; i++) {
      printf("%d. %s\n", i, argv[i]);
    }

    return (EXIT_SUCCESS);
}
```

y la ejecución luce como:

```
$ ./argumentos_main Hola Mundo Cruel
El numero de argumentos es 4
0. ./argumentos_main
1. Hola
2. Mundo
3. Cruel
```

5.5. Diseño top–down

Como ejemplo del diseño top–down haremos la implementación de la función seno utilizando serie de Taylor. La función seno puede ser calculada utilizando la serie de Taylor como

$$seno(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - ....$$

Para llevar a cabo la implementación necesitamos de dos funciones mas, la función factorial y la función potencia. Comenzaremos por escribir la función factorial. El factorial de un número se define como

$$1! = 1$$

$$2! = 1 * 2$$

$$3! = 1 * 2 * 3$$

$$4! = 1 * 2 * 3 * 4$$

$$n! = 1 * 2 * 3 * 4 * 5 * ... * n - 1 * n$$

y la implementación queda como

```
double factorial(int x)
{
  int i;
  double fact = 1;

  for(i=1; i<=x; i++)
    fact *= i;

  return fact;
}
```

Tenemos que elevar un número real a una potencia entera, así pues, si x es el número real para elevarlo a cualquier potencia hacemos:

$$x^2 = x * x$$

$$x^3 = x * x * x$$

$$x^n = x * x * x * ... * x$$

La implementación de esta función es:

```
double pow(double x, int n)
{
  int i;
  double pow =1;

  if(x==0) return 0;

  for(i=1; i<=n; i++)
    pow = pow*x;

  return pow;
}
```

En este caso escribiremos dos archivos, el primero Mat_fun.c y uno segundo ejemplo_uso_Mat_fun.c. El código del primero es

```
//Ejemplo de Mat_fun.c
/**
 * <p>Title: Funciones Trigonometricas</p>
 *
 * <p>Description: Implementa la función seno utilizando series de Taylor</p>
 *
 * <p>Copyright: Copyright (c) 2003</p>
 *
 * <p>Company: UMSNH</p>
 *
 * @author Dr. Felix Calderon Solorio
 * @version 1.0
 */

double seno(double x);
double factorial(int x);
double pow(double x, int n);


double seno(double x) {
  int i;
  double s = 0;
  int signo = 1;

  for (i = 1; i < 10; i += 2) {
    s += signo * pow(x, i) / factorial(i);
    signo *= -1;
  }
  return s;
}

double factorial(int x) {
  int i;
  double fact = 1;

  for (i = 1; i <= x; i++)
    fact *= i;

  return fact;
}
```

```
double pow(double x, int n) {
  int i;
  double pow = 1;

  if (x == 0)return 0;

  for (i = 1; i <= n; i++)
    pow = pow * x;

  return pow;
}
```

Para probar el archivo tenemos que escribir el código:

```
//ejemplo_uso_Mat_fun.c
/**
 * <p>Title: Uso de la funciones Mat_fun</p>
 *
 * <p>Description: Utiliza las funciones creadas </p>
 *
 * <p>Copyright: Copyright (c) 2003</p>
 *
 * <p>Company: UMSNH</p>
 *
 * @author Dr. Felix Calderon Solorio
 * @version 1.0
 */

#include <stdio.h>
#include "Mat_fun.c"

int main()
{
  int i,N;

  double inicio = -3.1416, fin = 3.1416, incremento = 0.1;

  N = (int)((fin - inicio)/incremento) + 1;

  double x, y;

  for(i=0;  i<N; i++)
```

```
  {
    x = inicio + incremento *i;
    y = seno(x);

printf("%f %f \n", x, y);
  }

  return 0;
}
```

Note que para incluir las funciones en C se utiliza la instrucción

```
#include "Mat_fun.c"
```

la cual da a conocer al compilador en donde se localizan las funciones.

5.5.1. Ceros de una función

En el caso de una función lineal el calcular los ceros de una función, es una tarea relativamente fácil donde despejando la variable en cuestión podemos calcular el valor para el cual la función se hace cero. En el caso de una función no lineal el problema es mas complejo y es común utilizar el método de Newton Raphson.

Dado que la función que queremos resolver es no lineal el Método de Newton Raphson hace una linealización por medio de la serie de Taylor así

$$f(x) = f(x_0) + \frac{df(x_0)}{dx}(x - x_0) + \frac{1}{2}\frac{d^2 f(x_0)}{dx^2}(x - x_0)^2 + \cdots$$

Si despreciamos los términos de orden mayor a dos podemos hacer una aproximación lineal para nuestra función

$$f(x_0) + f'(x_0)(x - x_0) = 0$$

y despejamos el valor de x tenemos que

$$x = x_0 - \frac{f(x_0)}{f'(x_0)}$$

Si el valor calculado de x lo llamamos x_1 tenemos un procedimiento iterativo para calcular la solución. A este procedimiento se le conoce como el algoritmo de Newton Raphson para

el calculo de ceros de una función. La iteración del método se muestra en la siguiente ecuación

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

Así dado un valor inicial x_0 lo suficientemente cerca, podemos calcular las raíces de la función. El siguiente código muestra los ceros de la función coseno calculados a partir de un punto inicial $x_0 = 0{,}5$

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/*
 * Newton_Raphson.c
 */
float f(float x);
float df(float x);
float Newton_Raphson(float x0);
/**
 * Funcion a calcular sus raices
 */

float f(float x) {
    return cos(x);
}

/**
 * Derivada de la funcion
 */

float df(float x){
    return -sin(x);
}

/**
 * Esta funcion calcula los ceros de una funcion por el metodo de
 * Newton Raphson
 */
```

```
float Newton_Raphson(float x0){
    float x = x0, diferencia;

    do {
        diferencia = f(x)/df(x);
        printf("%f %f\n", x, diferencia);
        x = x - diferencia;
    } while(diferencia*diferencia > 0.0000001);

    return x;
}

int main(int argc, char** argv) {
    printf("%f\n", Newton_Raphson(0.5));

    return (0);
}
```

Al ejecutar el código se produce la siguiente sucesión y la solución es 1.570796

```
0.500000 -1.830488
2.330488 0.949864
1.380623 -0.192499
1.573123 0.002326
1.570796 0.000000
1.570796
```

5.5.2. Raiz Cuadrada

Podemos utilizar el método de Newton Raphson para realizar el calculo de la raíz cuadrada si planteamos el problema como calcular los ceros de la función $f(x) = x^2 - R$. En este caso la solución utilizando el método de Newton Raphson es

$$f(x) = x^2 - R$$

$$f'(x) = 2x$$

Sustituyendo tenemos

$$x_{k+1} = x_k - \frac{x_k^2 - R}{2x_k}$$

$$x_{k+1} = \frac{2x_k^2 - x_k^2 + R}{2x_k}$$

$$x_{k+1} = \frac{x_k^2 + R}{2x_k}$$

Como valor inicial supondremos $x_0 = R$ y el código en C para esta implementación se muestra a continuación

```
/*
 * File:   raiz_cuadrada.c
 * Author: felix
 *
 * Created on 12 de octubre de 2011, 11:53
 */

#include <stdio.h>
#include <stdlib.h>

/*
 *
 */

float Raiz(float R){
    float x0 = R, x1 = R;

    do {
        x0 = x1;
        x1 = (x0 *x0  + R)/(2.0*x0);

        printf("%f \n", x1 );
    } while((x1-x0)*(x1-x0) > 0.0000001);

    return x1;
}


int main(int argc, char** argv) {
    printf("%f\n", Raiz(144));


    return (0);
```

```
}
```

Cuando corremos el código calculamos la raíz cuadrada de 144 y el resultado en la siguiente sucesión donde podemos notar que el valor final es 12, el cual corresponde a la raíz cuadrada de 144.

```
72.500000
37.243103
20.554794
13.780230
12.114991
12.000546
12.000000
12.000000
12.000000
```

5.5.3. Máximo Común Divisor

En matemáticas, se define el máximo común divisor (abreviado MCD) de dos o más números enteros al mayor número que los divide sin dejar resto. Por ejemplo, el MCD de 42 y 56 es 14. En efecto, $\frac{42}{14} = 3$, $\frac{56}{14} = 4$, y 3 y 4 son primos entre sí (no existe ningún número natural aparte de 1 que divida a la vez al 3 y al 4).

Al dividir a entre b (números enteros), se obtiene un cociente q y un residuo r. Es posible demostrar que el máximo común divisor de a y b es el mismo que el de b y r (Sea c el máximo común divisor de a y b. Como $a = bq + r$ y c divide a a y a b divide también a r. Si existiera otro número mayor que c que divide a b y a q, también dividiría a a , por lo que c no sería el mcd de a y b, lo que contradice la hipótesis). éste es el fundamento principal del algoritmo. También es importante tener en cuenta que el máximo común divisor de cualquier número a y 0 es precisamente a. Para fines prácticos, la notación $\text{mcd}(a, b)$ significa máximo común divisor de a y b.

Según lo antes mencionado, para calcular el máximo común divisor de 2366 y 273 se puede proseguir de la siguiente manera:

Paso	Operación	Significado
1	2366 dividido entre 273 es 8 y sobran 182	$\text{mcd}(2366, 273) = \text{mcd}(273, 182)$
2	273 dividido entre 182 es 1 y sobran 91	$\text{mcd}(273, 182) = \text{mcd}(182, 91)$
3	182 dividido entre 91 es 2 y sobra 0	$\text{mcd}(182, 91) = \text{mcd}(91, 0)$

La codificación del algoritmo de Euclides es:

```
/*
 * File:   MCD_MCM.c
```

```
 * Author: felix
 *
 * Created on 19 de septiembre de 2011, 17:05
 */

#include <stdio.h>
#include <stdlib.h>

/*
 *
 */

int EuclidesMCD(int a, int b){

    int aux;
    int mayor = a > b ? a : b;
    int menor = a < b ? a : b;

    do{
        aux = menor;
        menor = mayor % menor;
        mayor = aux;
    } while(menor!=0);
    return mayor;
}

int EuclidesMCM(int a, int b) {
    return(a/EuclidesMCD(a,b))*b;
}

int main(int argc, char** argv) {
    //int a = 10, b = 25;
    int a = 48, b= 60;
    printf("El Maximo comum Divisior de %d y %d es %d\n", a, b, EuclidesMCD(a, b));
    printf("El Minimo comum multiplo de %d y %d es %d\n", a, b, EuclidesMCM(a, b));

    return (EXIT_SUCCESS);
}
```

La manera en que calcularemos el MCD es utilizando el algoritmo de Euclides el cual se describe como: Entrada: Valores a y b pertenecientes a un dominio euclídeo

Salida: Un máximo común divisor de a y b
$r_0 \leftarrow a, r_1 \leftarrow b$
$i \leftarrow 1$
Mientras $r_i \neq 0$ haga lo siguiente: {
$r_{i+1} \leftarrow r_{i-1}$ mód r_i
$i \leftarrow i + 1$
}
El resultado es: r_{i-1}

5.6. Recursividad

Un método recursivo es un método que se llama a si mismo directa o indirectamente o a través de otro método. Un ejemplo interesante de recursion es la función factorial. La definición de factorial esta dada por el producto:

$$n! = n * (n-1) * (n-2) * ... * 3 * 2 * 1$$

Note que esta definición la podemos escribir de manera recursiva como:

$$n! = n * (n-1)!$$

En común representar a las funciones recursivas en dos pasos. El primero de ellos es conocido como Paso Base y Paso Recursivo. El primero es la condición más sencilla de la definición y normalmente es trivial hacer el cálculo y el segundo es el paso recursivo. Para el caso del Factorial tenemos

$factorial(N)$

Base

Si $N = 1$ el factorial es 1

Paso Recusivo

Si no $n \times factorial(N-1)$

Este método lo podemos escribir C queda como

```
// factorial.c

/**
```

```
 * <p>Title: factorial</p>
 *
 * <p>Description: Calcula el factorial de un numero usando recursividad</p>
 *
 * <p>Copyright: Copyright (c) 2003</p>
 *
 * <p>Company: UMSNH</p>
 *
 * @author Dr. Felix Calderon Solorio
 * @version 1.0
 */

#include <stdio.h>
double factorial(int N) ;

int main()
{
  int i;
  for (i = 0; i < 10; i++)
    printf("%d! = %f\n", i, factorial(i));

  return 0;
}

double factorial(int N)
{
  if (N == 0)return 1;
  else return (N * factorial(N - 1));
}
```

Otro ejemplo de definición recursiva es la multiplicación de números naturales. El producto a*b, donde a y b son enteros positivos, puede definirse como

$Multiplica(x, y)$

Base

Si $y = 1$ el producto es y

Paso Recusivo

Si no el resultado es $x + Multiplica(x, y - 1)$

```
// Ejemplo multiplica.c
```

```
/**
 * <p>Title: Multiplicación</p>
 *
 * <p>Description: Multiplica dos numeros utilizando recursividad</p>
 *
 * <p>Copyright: Copyright (c) 2003</p>
 *
 * <p>Company: UMSNH</p>
 *
 * @author Dr. Felix Calderon Solorio
 * @version 1.0
 */

#include <stdio.h>

double multiplica(int x, int y);

int main()
{
  int x=4, y=5;
  printf("%f\n", multiplica(4,5));
  return 0;
}

double multiplica(int x, int y)
{
   if(y == 1) return x;
   else return (x + multiplica(x,y-1));
}
```

5.6.1. Método de Bisecciones

Este método es conocido también como de corte binario. de partición en dos intervalos iguales o método de Bolzano. Es un método de búsqueda incremental donde el intervalo de búsqueda se divide en dos. Si la función cambia de signo sobre un intervalo, se calcula el valor en el punto medio. El nuevo intervalo de búsqueda será aquel donde el producto de la función cambie de signo.

La definición recursiva para este algoritmos es:

Dada una función $f(x)$ y un intervalo $[inicio, fin]$ de búsqueda donde existe un cruce por cero, podemos notar que $f(a) \times f(b) < 0$ porque la función en el intervalo pasa de menos a mas o de mas a menos debido a que cruza por cero. Así tenemos:

$Biseccion(inicio, fin)$

Base

Si $b - a < \epsilon$ la solución es $(a + b)/2$

Paso Recusivo

Calcular $mitad = \frac{inicio+fin}{2}$

Si $f(inicio) \times f(mitad) < 0$ entonces $Biseccion(inicio, mitad)$

Si no $Biseccion(mitad, fin)$

Ejemplo

Considere la función f(x) = x-cosx, a priori sabemos que la función tiene un cruce por cero en el intervalo [0,1], asÌ que nuestra búsqueda se concentrará en este.

iter	inicio	mitad	fin	f(ini)	f(mitad)	f(fin)
0	0.0	0.5	1.0	-1.0	-0.3775	0.4596
1	0.5	0.75	1.0	-0.3775	0.0183	0.4596
2	0.5	0.625	0.75	-0.3775	-0.1859	0.0183
3	0.625	0.6875	0.75	-0.1859	-0.0853	0.0183
4	0.6875	0.71875	0.75	-0.0853	-0.0338	0.0183
5	0.71875	0.734375	0.75	-0.0338	-0.0078	0.0183
6	0.734375	0.7421875	0.75	-0.0078	0.0051	0.0183
7	0.734375	0.73828125	0.7421875	-0.0078	-0.0013	0.0051
8	0.73828125	0.740234375	0.7421875	-0.0013	0.0019	0.0051
9	0.73828125	0.7392578125	0.740234375	-0.0013	0.0002	0.0019

La implementación recursiva de este algoritmo en C es:

```
//Ejemplo bisecciones.c

/**
 * <p>Title: Metodo de Biseccciones</p>
 *
 * <p>Description: Calcula el cruce por cero de una función utlizando el
 * metodo de biseccciones</p>
 *
 * <p>Copyright: Copyright (c) 2003</p>
 *
```

```
 * <p>Company: UMSNH</p>
 *
 * @author Dr. Felix Calderon Solorio
 * @version 1.0
 */

#include <stdio.h>
#include <Math.h>

double Biseccion(double ini, double fin);
double funcion(double x);

int main()
{
  printf("La solucion esta en %f\n", Biseccion(0,1));
  return 0;
}

double Biseccion(double ini, double fin)
{
  double mitad;    mitad = (fin + ini)/2.0;
  printf("f_1(%f) = %f f_m(%f) = %f f_2(%f) = %f\n", ini, funcion(ini),
          mitad, funcion(mitad), fin, funcion(fin));
  if((fin - ini) > 1e-12)
  {
    if(funcion(ini)*funcion(mitad) < 0)
      return Biseccion(ini, mitad);
    else return Biseccion(mitad, fin);
  }
  else return (mitad);
}

double funcion(double x)
{
  return (x - cos(x));
}
```

5.6.2. Torres de Hanoi

El ejemplo más conocido, de recursividad entre los programadores, es el problema de las torres de Hanoi. Para este juego, se tienen tres postes (A, B, C) y un conjunto n de discos colocados de mayor a menor diámetro en alguno de los postes, por ejemplo el poste A. La meta es mover todos los discos del poste A al poste C con las siguientes restricciones: en cada movimiento solo se puede tomar un disco y colocarlo en cualquier poste y no se debe colocar, un en un poste, un disco de diámetro mayor sobre un disco de diámetro menor.

La definición recursiva de las torres de Hanoi es:

$mover\_discos(origen, auxiliar, destino, discos)$

Base

Si $discos = 1$ Mover un disco de *origen* a *destino*

Paso recursivo

Mover $n - 1$ discos de *origen* a *auxiliar*

Mover 1 disco de *origen* a *destino*

Mover $n - 1$ discos de *auxiliar* a *destino*

La implementación en C queda

```
/*
 * File:   torres_hanoi.c
 * Author: felix
 *
 * Created on 18 de octubre de 2011, 10:38
 */

#include <stdio.h>
#include <stdlib.h>

/*
 * Torres de Hanoi
 */

int movtos ;
void mover_discos(char a, char b, char c, int n);

int main()
```

```
{
    movtos = 1;
   mover_discos('A', 'B', 'C', 4);
   return 0;
}

void mover_discos(char a, char b, char c, int n)
{
    if (n > 0) {
        mover_discos(a, c, b, n - 1);
        printf("%d Mover un disco del poste %c al poste %c\n", movtos++, a, c);
        mover_discos(b, a, c, n - 1);
    }
}
```

La solución con tres discos es:

```
Mover un disco del poste A al poste B

Mover un disco del poste A al poste C

Mover un disco del poste B al poste C

Mover un disco del poste A al poste B

Mover un disco del poste C al poste A

Mover un disco del poste C al poste B

Mover un disco del poste A al poste B
```

5.6.3. Algoritmo de quicksort

Sea x un arreglo y n el número de elementos en arreglo que se debe ordenar. Elegir un elemento a de una posición especifica en el arreglo (por ejemplo, a puede elegirse como el primer elemento del arreglo. Suponer que los elemento de x están separados de manera que a está colocado en la posición j y se cumplen las siguientes condiciones.

1. Cada uno de los elementos en las posiciones de 0 a j-1 es menor o igual que a.
2. Cada uno de los elementos en las posiciones j+1 a n-1 es mayor o igual que a.

Observe que si se cumplen esas dos condiciones para una a y j particulares, a es el j-ésimo menor elemento de x, de manera que a se mantiene en su posición j cuando el arreglo está ordenado en su totalidad. Si se repite este procedimiento con los subarreglos que van de x[0] a x[j-1] y de x[j+1] a x[n-1] y con todos los subarreglos creados mediante este proceso, el resultado final será un archivo ordenado.

Ilustremos el quicksort con un ejemplo. Si un arreglo esta dado por:

x = [25 57 48 37 12 92 86 33]

y el primer elemento se coloca en su posición correcta, el arreglo resultante es:

x = [12 25 57 48 37 92 86 33]

En este punto 25 esta en su posición correcta por lo cual podemos dividir el arreglo en

x = [12] 25 [57 48 37 92 86 33]

Ahora repetimos el procedimiento con los dos subarreglos

x = 12 25 [48 37 33] 57 [92 86]

x = 12 25 33 [37 48] 57 [86] [92]

x = 12 25 33 [37 48] 57 86 92

x = 12 25 33 37 48 57 86 92

El procedimiento es entonces.

Buscar la partición del arreglo j. Ordenar el sub-arreglo x[0] a x[j-1] Ordenar el sub-arreglo x[j+1] a x[n-1]

Su implementación en Java es:

```
public void quiksort(int x[],int lo,int ho)
{
  int t, l=lo, h=ho, mid;

  if(ho>lo)
  {
    mid=x[(lo+ho)/2];
    while(l<=h)
    {
      while((l<ho)&&(x[l]<mid))  ++l;
      while((h>lo)&&(x[h]>mid))  --h;
      if(l<=h)
      {
```

```
          t    = x[l];
          x[l] = x[h];
          x[h] = t;
          ++l;
          --h;
        }
      }

      if(lo<h) quiksort(x,lo,h);
      if(l<ho) quiksort(x,l,ho);
    }
  }
```

Arreglos y Punteros

6.1. Introducción

Un apuntador o Puntero es una variable que contiene la dirección en memoria de otra variable. Se pueden tener apuntadores a cualquier tipo de variable. La manera de declarar un apuntador es utilizando * así por ejemplo:

char * a;	es un apuntador a una variable de tipo char
int *a;	es un apuntador a entero
float *a;	es un apuntador a una variable de tipo float
double *a;	es un apuntador a una variable tipo double
tipo *a	es un apuntador a una variable de tipo general

Así por ejemplo cuando se tenga la declaración

```
void main() {
      int *a;
      char *b;
}
```

Las variables a y b son apuntadores y representan la dirección física en memoria de donde se almacena los valores en cada variable.

Un apuntador es la variable mas versátil de C ya que podemos acceder el contenido de cualquier localidad de memoria y cambiar su contenido. Esta posibilidad le da a C una capacidad para cambiar los contenidos de video, puertos, impresoras, tarjetas de video, variables, etc.

Los apuntadores tendrán un doble significado, así por ejemplo

```
void main() {
int *a = 10;
printf("%X, %d\n", a, *a);
}
```

a	representa la dirección de memoria donde se almacena el número 10
*a	representa el número 10 almacenado en memoria

6.2. Direcciones de variables

El operador unario o monádico & devuelve la dirección de memoria de una variable, a diferencia del operador de indirección o deferencia * devuelve el "contenido de un objeto apuntado por un apuntador".

Para declarar un apuntador para una variable entera hacer:

```
int *apuntador;
```

Se debe asociar a cada apuntador un tipo particular. Por ejemplo, no se puede asignar la dirección de un short int a un long int.

Para tener una mejor idea, considerar el siguiente código:

```
 main() {
    int x = 1, y = 2;
    int *ap;

    ap = &x;

    y = *ap;

    x = ap;

    *ap = 3;
}
```

Con el objetivo de entender el comportamiento del código supongamos que la variable x esta en la localidad de la memoria 100, y en 200 y ap en 1000. Nota: un apuntador es una variable, por lo tanto, sus valores necesitan ser guardados en algún lado.

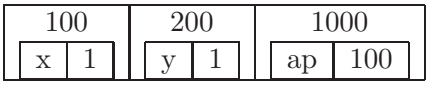
```
   int x = 1, y = 2;
   int *ap;

   ap = &x;
```

100		200		1000	
x	1	y	2	ap	100

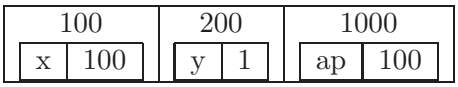
Las variables x e y son declaradas e inicializadas con 1 y 2 respectivamente, ap es declarado como un apuntador a entero y se le asigna la dirección de x (&x). Por lo que ap se carga con el valor 100.

y = *ap;



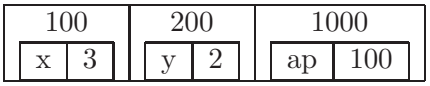
Después y obtiene el contenido de ap. En el ejemplo ap apunta a la localidad de memoria 100 – la localidad de x. Por lo tanto, y obtiene el valor de x – el cual es 1.

x = ap;



Como se ha visto C no es muy estricto en la asignación de valores de diferente tipo (apuntador a entero). Así que es perfectamente legal (aunque el compilador genera un aviso de cuidado) asigna el valor actual de ap a la variable x. El valor de ap en ese momento es 100.

*ap = 3;



Finalmente se asigna un valor al contenido de un apuntador (*ap).

Importante: Cuando un apuntador es declarado apunta a algún lado. Se debe inicializar el apuntador antes de usarlo. Por lo que:

main() int *ap; *ap = 100; puede generar un error en tiempo de ejecución o presentar un comportamiento errático.

El uso correcto será:

```
main() {
   int *ap;
   int x;

   ap = &x;
   *ap = 100;
}
```

Con los apuntadores se puede realizar también aritmética entera, por ejemplo:

```
main() {
```

```
    float *flp, *flq;

    *flp = *flp + 10;

    ++*flp;

    (*flp)++;

    flq = flp;
}
```

NOTA: Un apuntador a cualquier tipo de variables es una dirección en memoria – la cual es una dirección entera, pero un apuntador NO es un entero.

La razón por la cual se asocia un apuntador a un tipo de dato, es por que se debe conocer en cuantos bytes esta guardado el dato. De tal forma, que cuando se incrementa un apuntador, se incrementa el apuntador por un "bloque" de memoria, en donde el bloque esta en función del tamaño del dato.

Por lo tanto para un apuntador a un char, se agrega un byte a la dirección y para un apuntador a entero o a flotante se agregan 4 bytes. De esta forma si a un apuntador a flotante se le suman 2, el apuntador entonces se mueve dos posiciones float que equivalen a 8 bytes.

El siguiente es un ejemplo de como a través de un apuntador podemos cambiar el contenido de una variable.

```
/*
 * File:   cambia_valores.c
 * Author: felix
 *
 * Created on 25 de octubre de 2011, 9:46
 */

#include <stdio.h>
#include <stdlib.h>

/*
 * Ejemplo de cambio de valores
 */
int main(int argc, char** argv) {
    int a = 10, b = 15, c = 20;
    int *p;
```

```
    printf("a = %d b = %d c = %d\n", a, b, c);
    printf("direcciones de a = %X b = %X y C = %X\n", &a, &b, &c);

    p = &a;
    *p = 15;

    p = &b;

    *p = 16;

    printf("a = %d b = %d c = %d\n", a, b, c);

    printf("En la direccion %X hay un %d\n", p, *p);
    p++;
    printf("En la direccion %X hay un %d\n", p, *p);

    return (0);
}
```

6.3. Punteros como argumentos

Los punteros puwden ser utilizados como argumentos de una función. El caso mas conocido y que hemos estado utilizando es la función scanf. En esta función cuando hacemos el llamado a ella escribimos

```
float a ;
scanf("%f", &a);
```

En este caso la función scanf, recibe dos parametros, una cadena de control y la dirección donde se almacena la variable a. El prototipo de la función scanf es:

```
scanf(char *, float *);
```

Una de las aplicaciones mas interesante de los apuntadores es cuando queremos que en una función los argumentos sean modificados dentro del cuerpo de la función. Un ejemplo de esto es la función inc del siguiente ejemplo

```
/*
 * File:   incremento.c
 * Author: felix
 *
```

```
 * Created on 27 de octubre de 2011, 9:37
 */

#include <stdio.h>
#include <stdlib.h>

/*
 *
 */
void inc01(int x) {
    x ++;
}

int inc(int x) {
    x++;
    return x;
}

void inc2(int *x) {
    (*x)++;
}

int main(int argc, char** argv) {
    int x = 10;
    printf("x = %d\n", x);
    inc01(x);
    printf("x = %d\n", x);
    x = inc(x);
    printf("x = %d\n", x);
    inc2(&x);
    printf("x = %d\n", x);
    return (EXIT_SUCCESS);
}
```

6.4. Paso por valor y por referencia en funciones

En C el paso por valor significa que al compilar la función y el código que llama a la función, Esta recibe una copia de los valores de los parámetros que se le pasan como argumentos.

Las variables reales no se pasan a la función, solo copias de su valor.

Cuando una función debe modificar el valor de la variable pasada como parámetro y que esta modificación retorne a la función llamadora, se debe pasar el parámetro por referencia. En este método, el compilador no pasa una copia del valor del argumento; en su lugar, pasa una referencia, que indica a la función donde existe la variable en memoria.

La referencia que una función recibe es la dirección de la variable. Es decir, pasar un argumento por referencia es, simplemente, indicarle al compilador que pase la dirección del argumento.

Ejemplo:

```
void demo(int &valor) {
  valor=5;
  printf("%d\n",valor);
}

void main() {
  int n=10;
  printf("%d\n", n);
  demo(n);
  printf("%d\n", n);
}
```

La salida de este programa será 5;

Ejemplo de paso por referencia

Dados dos números a y b hacer un programa que reciba como argumentos estos dos números y que independientemente de los valores siempre en la variable a este el mayor y en b el menor. La implementación en C es:

```
//Ejemplo paso_x_referencia.c

/**
 * <p>Title: Paso por referencia</p>
 *
 * <p>Description: Cambia el valor de dos numeros en una función</p>
 *
 * <p>Copyright: Copyright (c) 2003</p>
 *
 * <p>Company: UMSNH</p>
```

```
 *
 * @author Dr. Felix Calderon Solorio
 * @version 1.0
 */

#include <stdio.h>

void cambia(float *x, float *y);

int main() {

  float a = 2;
  float b = 3;

  printf("Antes   %f %f\n", a, b);

  cambia(&a, &b);

  printf("Despues %f %f\n", a, b);

  return 0;
}

void cambia(float *x, float *y)
{
  float temp = *x;
  *x = *y;
  *y = temp;

  return;
}
```

Cuando en C se utiliza &a, se esta pasando la dirección de la variable a, en la memoria donde reside el dato; y cuando *a es el apuntador de la memoria (apuntadores será revisado en los siguientes capítulos).

6.5. Aritmética de apuntadores

Como ya se mencionó los apuntadores tienen asociado un tipo de datos, este es muy importante cuando necesitamos realizar operaciones con un apuntador. En esta sección ana-

lizaremos la aritmética de apuntadores.

Sea:

```
int a[10], *ap;
ap = a; /*dirección del primer elemento */
```

a es el nombre de un arreglo de enteros, recuerda que el nombre de un arreglo se define como:

a ≡ & a[0]

por lo tanto el nombre de cualquier arreglo es un APUNTADOR CONSTANTE, esto implica que:

ap ≡ & a[0]

El siguiente esquema muestra la relación existente entre los elementos del arreglo a y del apuntador ap:

a =	a[0]	a[1]	a[2]	...	a[n-1]
	0008	___	___	....	___
ap =	1000	1004	1008	...	4*(n-1)+1000

Como se puede observar el apuntador ap más un número entero, equivale a la dirección de alguno de los elementos del arreglo, esto es:

(ap + 0) ≡ & a[0]

(ap + 1) ≡ & a[1]

....

(ap + 8) ≡ & a[8]

(ap + 2) ≡ & a[0] + (2 * sizeof(int))

ap ++; /* apunta a & a[1] */

ap += 3; /* apunta a & a[4] */

De la misma manera el contenido del apuntador hace referencia al contenido de alguno de los elementos del arreglo, es decir:

```
*( ap + 0 )  a[0]

*( ap + 1 )  a[1]
```

```
.... ...

*( ap + 8 )  a[8]


*( ap + 2 )  *(a[0] + ( 2 * sizeof(int) ) )

*(ap ++); /* contenido del elemento a[1] */

*(ap += 3); /* contenido del elemento a[4] */
```

El nombre de un arreglo es un apuntador al primer elemento del mismo,1 por eso basta con asignar el nombre a un apuntador, para que este apunte al inicio del arreglo.

Cuando le sumamos un entero a un apuntador, este avanza tantas direcciones como bytes ocupen el tipo de dato del arreglo. En el ejemplo de arriba nota que los apuntadores se incrementan de 4 en 42

La resta de apuntadores es valida siempre y cuando ambos apunten a un mismo arreglo, el resultado es el numero de celda que existen entre las celdas apuntadas. Si restamos bp-ap obtenemos 93

Los apuntadores pueden compararse entre sí, siempre y cuando apunten a un mismo arreglo. Las comparaciones se hacen en base a la posición de las celdas apuntadas. Por ejemplo:

```
if (ap == bp) /* es falsa */

if (ap != bp ) /* es cierta */

if (ap > bp ) /* es falsa */

if (ap < bp ) /* es cierta */

if (ap >= bp ) /* es falsa */

if (ap <= bp ) /* es cierta */
```

El siguiente ejemplo muestra como se pueden acceder localidades de memoria consecutivas y colocar un valor e imprimirlo.

```
Cualquier otra operación con apuntadores NO ES VALIDA.
/*
```

```
 * File:   aritmetica.c
 * Author: felix
 *
 * Created on 5 de mayo de 2012, 19:14
 */

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    int a[10];
    int *p, i;
    p = a;

    for(i=0; i<10; i++)
        *(p + i) = i+1;

    for(i=0; i<10; i++)
        printf("%D %d\n" , (p+i), *(p + i));

    return (EXIT_SUCCESS);
}
```

Como resultado de la ejecución podemos ver que un incremento unitario equivale a 4 en dirección de memoria, lo cual corresponde al tamaño de un entero.

6.6. Definición de arreglo y declaración tipo vector y tipo matriz

Los vectores y matrices en matemáticas se definen como arreglos de números representados por una variable. Los vectores se representaran como

$$a = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$$

y las matrices se representan como

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,N} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,N} \\ \cdots & \cdots & \cdots & \vdots \\ a_{M,1} & a_{M,2} & \cdots & a_{M,N} \end{pmatrix}$$

6.6.1. Los vectores

Imaginemos que tenemos que hallar el promedio de 10 números que introduzca el usuario (o realizar cualquier otra operación con ellos). Parece evidente que tiene que haber una solución más cómoda que definir 10 variables distintas y escribir 10 veces las instrucciones de avisar al usuario, leer los datos que teclee, y almacenar esos datos. Si necesitamos manejar 100, 1000 o 10000 datos, resulta todavía más claro que no es eficiente utilizar una variable para cada uno de esos datos.

Por eso se emplean los arrays (o arreglos). Un array es una variable que puede contener varios dato del mismo tipo. Para acceder a cada uno de esos datos emplearemos corchetes. Por ejemplo, si definimos una variable llamada m que contenga 10 números enteros, accederemos al primero de estos números como m[0], el último como m[9] y el quinto como m[4] (se empieza a numerar a desde 0 y se termina en n-1). Veamos un ejemplo que halla la media de cinco números (con decimales, "double"):

```
/*
 * File:   Arreglos.c
 * Author: felix
 *
 * Created on 27 de octubre de 2011, 9:18
 */

#include <stdio.h>
#include <stdlib.h>

/*
 *
 */

void ejemplo01(){
    float a[10];
    int i;

    // Lee 10 numeros de teclado
```

```
    for(i=0; i<10; i++) {
        printf("Dame el valor a[%d] = ", i);
        scanf("%f", &a[i]);
    }


    // Imprime 10 numeros

    for(i=0; i<10; i++)
        printf("%f\n", a[i]);
}

void ejemplo02(){
    float a[10];
    int i;

    // Lee 10 numeros de teclado

    for(i=0; i<10; i++) {
        printf("Dame el valor a[%d] = ", i);
        scanf("%f", a+i);
    }


    // Imprime 10 numeros

    for(i=0; i<10; i++)
        printf("%f\n", *(a+i));
}

int main(int argc, char** argv) {

    ejemplo01();
    return (EXIT_SUCCESS);
}
```

Para definir la variable podemos usar dos formatos: “float a[10]” que es la sintaxis habitual en C en el cual se declaran los arreglos de manera fija a tiempo de ejecución o de manera dinámica.

Lo habitual no será conocer los valores en el momento de teclear el programa, como hemos

hecho esta vez. Será mucho más frecuente que los datos los teclee el usuario o bien que los leamos de algún fichero, los calculemos, etc. En este caso, tendremos que reservar el espacio, y los valores los almacenaremos a medida que vayamos conociéndolos.

Para ello, primero declararíamos que vamos a utilizar un array, así:

```
float  *datos;
```

y después reservaríamos espacio (por ejemplo, para 1000 datos) con

```
datos = (float *) malloc(1000*sizeof(float));
```

o también

```
datos = (float *) calloc(1000, sizeof(float));
```

Los valores los podemos asignar de una forma similar a la que hemos visto en el ejemplo anterior:

```
datos[25] = 100 ; datos[0] = i*5 ; datos[j+1] = (j+5);
```

Vamos a ver un ejemplo algo más completo, con arrays de números enteros, llamados a, b y c. A uno de ellos (a) le daremos valores al definirlo, otro lo definiremos en dos pasos (b) y le daremos fijos, y el otro lo definiremos en un paso y le daremos valores calculados a partir de ciertas operaciones:

La implementación en C es

```
//inicia_arreglo.c

/**
 * <p>Title: Arreglos</p>
 *
 * <p>Description: inicializa arreglos de varias formas</p>
 *
 * <p>Copyright: Copyright (c) 2003</p>
 *
 * <p>Company: UMSNH</p>
 *
 * @author Dr. Félix Calderon Solorio
 * @version 1.0
 */

#include <stdio.h>
#include <stdlib.h>
```

```
int main( ) {

     int i;  // Para repetir con bucles "for"

     // ------  Primer array de ejemplo
     int a[] = { 10, 12345, -15, 0, 7 };

     printf( "Los valores de a son:\n" );
     for (i=0; i<5; i++)
       printf("%d\n", a[i]);

     // ------  Segundo array de ejemplo

 int *b  = (int*) calloc(3, sizeof(int));

     b[0] = 15; b[1] = 132; b[2] = -1;

     printf( "Los valores de b son:\n" );
     for (i=0; i<3; i++)
       printf("%d\n", b[i] );

     // ------  Tercer array de ejemplo
     int j = 4;
     int *c = new int[j];

     for (i=0; i<j; i++)
       c[i] = (i+1)*(i+1);

     printf("Los valores de c son: \n" );
     for (i=0; i<j; i++)
       printf("%d\n", c[i]);

 return 0;

}
```

En general la forma mas común y que permite el manejo dinámico de memoria es utilizando la instrucción calloc o malloc. Esta la podemos escribir dentro de una función a la denominaremos vector, dicha función recibe el tamaño del arreglo y regresa un apuntador con la dirección a partir de la cual se guardaron los N valores. En el siguiente código se muestra esto

```
/*
 * File:   vector.c
 * Author: felix
 *
 * Created on 16 de mayo de 2012, 5:22
 */

#include <stdio.h>
#include <stdlib.h>

/*
 *
 */

float * vector(int N) {

    float *ap;

    ap = (float *) calloc(N, sizeof(float));

    return ap;
}

int main(int argc, char** argv) {

    float *v;

    v= vector(10);

    return (EXIT_SUCCESS);
}
```

En este caso cuando se llama a la función vector(10) regresa un apuntador con 10 reservaciones de flotantes numerados del 0 al 9 respectivamente. Para hacer una reservación de diferente tipo solamente hay que cambiar float por cualquier otro tipo en todas las variables involucradas en la función.

El siguiente código es un ejemplo de un arreglo que es inicializando con los números de 1 al 10 y como imprimir los números contenidos en el.

```
/*
 * File:   vector.c
```

```
 * Author: felix
 *
 * Created on 16 de mayo de 2012, 5:22
 */

#include <stdio.h>
#include <stdlib.h>

float * vector(int N) {
    float *ap;
    ap = (float *) calloc(N, sizeof(float));
    return ap;
}

void llena(float *a, int N) {
    int i;

    for(i=0; i<N; i++)
        a[i] = i+1;
}

void imprime(float *a, int N) {
    int i;

    for(i=0; i<N; i++)
        printf("%f\n", a[i]);
}

int main(int argc, char** argv) {

    float *v;

    v= vector(10);
    llena(v, 10);
    imprime(v, 10);
    return (EXIT_SUCCESS);
}
```

Ejercicio

Hacer una función que regrese, los primeros 10 números de la sucesión de Fibonacci.

6.6.2. Matrices

Las matrices se definen, como un arreglo bidimensional, en donde tenemos un número de renglones N y un número de columnas M. La representación matemática de una matriz es :

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,M} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,M} \\ \cdots & \cdots & \cdots & \cdots \\ a_{N,1} & a_{N,2} & \cdots & a_{N,M} \end{pmatrix}$$

Para hacer la definición de un arreglo bidimensional en C, se hace es mediante la siguiente instrucción

```
double a[][] = {{1,2,3}, {4,5,6}, {7,8,9}};
```

donde a es el nombre del arreglo 3 el número de renglones y 3 el número de columnas. Para hacer referencia al elemento en el i-esimo renglón y la j-esima columna hacemos a[i][j], por ejemplo para imprimir el valor el la posición a[0][0] escribimos

```
printf("%f\n", a[0][0]);
```

el arreglo bidemensional o matriz es

$$a = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

6.6.3. Ejemplo

Hacer un programa que almacene la matriz anterior en un arreglo,la eleve al cuadrado e imprima el resultado en la pantalla.

La implementación pero en C es

```
/*
 * File:   arreglo_bidimensional.c
 * Author: felix
 *
 * Created on 16 de mayo de 2012, 6:18
 */
```

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  double a[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};
  double b[3][3] = {0,0,0,0,0,0,0,0,0} ;

  int i, j;


  for(i=0; i< 3; i++)
    for(j=0; j<3; j++)
      b[i][j] = a[i][j] * a[i][j];

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

    return (EXIT_SUCCESS);
}
```

6.7. Manipulación de arreglos y punteros

Una forma alterna de declarar una matriz es utilizando las instrucciones calloc y malloc ya que de esta forma podemos declarar arreglos de cualquier longitud en tiempo de ejecución. La manera de declarar una matriz de flotantes es utilizando apuntadores, un * será un apuntador a un arreglo unidimensional. dos ** a un arreglo bidimensional y así sucesivamente. Para declara el arreglo bidimensional M escribimos:

```
float **M;
```

El siguiente paso es reservar memoria utilizando las instrucciones calloc, esto lo realizamos en dos pasos, primero hacemos la declaración de los renglones para después declarar las columnas

```
M = (TIPO **) calloc(ren, sizeof(TIPO*)); // buscar en stdlib.h
```

donde TIPO es cualquier tipo de datos declarado por el usuario. Posteriormente para cada columna reservamos la memoria haciendo

```
  for(i=0; i<ren; i++)
M[i] = (TIPO *) calloc(col, sizeof(TIPO));
```

finalmente tenemos un arreglo M de TIPO de tamaño $ren \times col$. Todo esto lo podemos ver en la función Matriz que podemos usar en cualquier otro código

```
TIPO ** Matriz(int ren, int col)
{
  TIPO **aux;
  int i, j;

  aux = (TIPO **) calloc(ren, sizeof(TIPO*)); // buscar en stdlib.h

  for(i=0; i<ren; i++)
aux[i] = (TIPO *) calloc(col, sizeof(TIPO));

  return aux;
}
```

6.7.1. Ejemplo

Hacer una función que reciba un número y regrese un cuadro de texto lleno de asteriscos, del mismo tamaño que el número que recibe y con el número en la diagonal. Así pues, esta función dado el número 4 regresara la información

```
4***
*4**
**4*
***4
```

La implementación en C queda

```
//Ejemplo ec043.cpp

/**
 * <p>Title: Ejemplo de llenado de matrices</p>
 *
 * <p>Description: Llena una matriz con asteriscos</p>
 *
 * <p>Copyright: Copyright (c) 2005</p>
```

```
 *
 * <p>Company: UMSNH</p>
 *
 * @author Dr. Felix Calderon Solorio
 * @version 1.0
 */

/*
 * File:   cuadro.c
 * Author: felix
 *
 * Created on 16 de mayo de 2012, 10:25
 */

#include <stdio.h>
#include <stdlib.h>

typedef char TIPO;

void imprime(char **a, int n);
char ** llena(int n);
char ** Matriz(int ren, int col);

int main() {
  char **a;
  int n = 5;

  a = llena(n);
  imprime(a, n);

  return 0;
}

void imprime(char **a, int n) {
  int i, j;

  for(i=0; i<n; i++)
  {
    for (j = 0; j < n; j++)
      printf("%c", a[i][j]);
    printf("\n");
```

```
  }
}


char ** llena(int n) {
  char **aux;
  int i, j;

  aux = Matriz(n, n);

  for(i=0; i<n; i++)
    for(j=0; j<n; j++)
      if(i==j) aux[i][j] = (char) (n+48);
      else aux[i][j] = '*';

  return aux;

}

TIPO ** Matriz(int ren, int col)
{
  TIPO **aux;
  int i, j;

  aux = (TIPO **) calloc(ren, sizeof(TIPO*)); // buscar en stdlib.h

  for(i=0; i<ren; i++)
aux[i] = (TIPO *) calloc(col, sizeof(TIPO));

  return aux;
}
```

6.8. Diferencia entre variables tipo apuntador y variables tipo arreglo

Existe una relación estrecha entre los punteros y los arreglos. En C, un nombre de un arreglo es un índice a la dirección de comienzo del arreglo. En esencia, el nombre de un arreglo es un puntero al arreglo. Considerar lo siguiente:

```
int a[10], x; int *ap;
```

```
ap = &a[0];    /* ap apunta a la direccion de a[0] */

x = *ap;       /* A x se le asigna el contenido de ap (a[0] en
este caso) */

*(ap + 1) = 100;  /* Se asigna al segundo elemento de 'a' el valor
100 usando ap*/
```

Como se puede observar en el ejemplo la sentencia a[t] es idéntica a ap+t. Se debe tener cuidado ya que C no hace una revisión de los límites del arreglo, por lo que se puede ir fácilmente mas allá del arreglo en memoria y sobreescribir otras cosas.

C sin embargo es mucho más sutil en su relación entre arreglos y apuntadores. Por ejemplo se puede teclear solamente:

ap = a; en vez de ap = &a[0]; y también *(a + i) en vez de a[i], esto es, &a[i] es equivalente con a+i. Y como se ve en el ejemplo, el direccionamiento de apuntadores se puede expresar como:

a[i] que es equivalente a *(ap + i) Sin embargo los apuntadores y los arreglos son diferentes:

Un apuntador es una variable. Se puede hacer ap = a y ap++. Un arreglo NO ES una variable. Hacer a = ap y a++ ES ILEGAL. Este parte es muy importante, asegúrese haberla entendido.

Con lo comentado se puede entender como los arreglos son pasados a las funciones. Cuando un arreglo es pasado a una función lo que en realidad se le esta pasando es la localidad de su elemento inicial en memoria.

Por lo tanto:

strlen(s) es equivalente a strlen(&s[0]) Esta es la razón por la cual se declara la función como:

int strlen(char s[]); y una declaración equivalente es int strlen(char *s); ya que char s[] es igual que char *s.

La función strlen() es una función de la biblioteca estándar que regresa la longitud de una cadena. Se muestra enseguida la versión de esta función que podría escribirse:

```
int strlen(char *s) {
    char *p = s;

    while ( *p != '\0' )
```

```
        p++;
    return p - s;
}
```

Se muestra enseguida una función para copiar una cadena en otra. Al igual que en el ejercicio anterior existe en la biblioteca estándar una función que hace lo mismo.

```
void strcpy(char *s, char *t) {
    while ( (*s++ = *t++) != '\0' );
}
```

En los dos últimos ejemplos se emplean apuntadores y asignación por valor. Nota: Se emplea el uso del carácter nulo con la sentencia while para encontrar el fin de la cadena.

6.9. Matrices, Operaciones básicas

Una matriz se define como un arreglo bidimensional de datos, que tiene n renglones y m columnas. Un elemento del arreglo puede ser identificado con a_{ij}

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & \dots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & a_{1,2} & \dots & a_{1,n-1} \\ a_{2,0} & a_{2,1} & a_{2,2} & \dots & a_{2,n-1} \\ \\ a_{m-1,0} & a_{m-1,1} & a_{m-1,2} & \dots & a_{m-1,n-1} \end{pmatrix}$$

Algunas de las operaciones básicas que pueden realizarse con matrices son suma, resta y multiplicación. La división de matrices como tal no existe y en su lugar se calcula la inversa.

6.9.1. Suma de matrices

Para que la sumar las matrices A y B, se requiere que las matrices tengan el mismo número de renglones y de columnas. Si queremos encontrar la suma C = A + B, cada elemento de la matriz C lo calculamos de la siguiente forma:

$$C_{i,j} = A_{i,j} + B_{i,j}$$

para todos lo i, j en la matriz C

6.9.2. Resta de matrices

En este caso, se deben cumplir las mismas propiedades que la resta de matrices y el calculo de los elemento de la matriz C se calculan como:

$$C_{i,j} = A_{i,j} - B_{i,j}$$

para todos lo i, j en la matriz C

6.9.3. Multiplicación de matrices

Para realizar el producto $C = A * B$ tenemos que considerar que el producto existe si

1.- El número de columnas de A es igual al número de renglones de B.

$$C(n, l) = A(n, m) * B(m, l)$$

2.- Las dimensiones de la matriz resultante tendrá el mismo numero de renglones que la matriz A y el mismo número de columnas que la matriz B.

3.- El cálculo de los elementos de la matriz C se lleva a cabo haciendo :

$$c_{i,j} = \sum_{k=0}^{m-1} a_{i,k} * b_{k,j}$$

para todos lo i, j en la matriz C

6.9.4. Solución de sistemas de ecuaciones triangulares

Considere el siguiente sistema de ecuaciones

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ 0 & a_{1,1} & a_{1,2} \\ 0 & 0 & a_{2,2} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix}$$

Podemos ver que la solución para x_2 es

$$x_2 = \frac{b_2}{a_{2,2}}$$

para x_1 es

$$x_1 = \frac{b_1 - a_{1,2}x_2}{a_{1,1}}$$

y para x_0

$$x_0 = \frac{b_0 - a_{0,1}x_1 + a_{0,2}x_2}{a_{0,0}}$$

en general tenemos que

$$x_k = \frac{b_k - \sum_{j=k+1}^{n-1} a_{k,j}x_j}{a_{k,k}}$$

Determinantes

Podemos ver que el determinante de una matriz triangular superior o inferior es :

$$det(A) = a_{0,0} * a_{1,1}a_{2,2} * ... * a_{n-1,n-1}$$

6.9.5. Solución de sistemas lineales de ecuaciones por el método de Eliminación Gaussiana

Consideremos que tenemos un sistema lineal Ax=b, donde la matriz A no tiene las condiciones de ser triangular superior.

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix}$$

Comenzaremos por despejar x_0 de la ecuación (I)

$$x_0 = \frac{b_0 - a_{0,1}x_1 - a_{0,2}x_2}{a_{0,0}}$$

y sustituimos en las ecuaciones (II)

$$a_{1,0}\frac{b_0 - a_{0,1}x_1 - a_{0,2}x_2}{a_{0,0}} + a_{1,1}x_1 + a_{1,2}x_2 = b_1$$

agrupando términos semejantes tenemos:

$$(a_{1,1} - \frac{a_{1,0} * a_{0,1}}{a_{0,0}})x_1 + (a_{1,2} - a_{1,0}\frac{a_{0,2}}{a_{0,0}})x_2 = (b_1 - a_{1,0}\frac{b_0}{a_{0,0}})$$

Ahora sustituimos en la ecuación (III)

$$a_{2,0}\frac{b_0 - a_{0,1}x_1 - a_{0,2}x_2}{a_{0,0}} + a_{2,1}x_1 + a_{2,2}x_2 = b_2$$

$$(a_{2,1} - a_{2,0}\frac{a_{0,1}}{a_{0,0}})x_1 + (a_{2,2} - a_{2,0}\frac{a_{0,2}}{a_{0,0}})x_2 = (b_2 - a_{2,0}\frac{b_0}{a_{0,0}})$$

Lo cual nos da un nuevo sistema simplificado dada por

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ 0 & a'_{1,1} & a'_{1,2} \\ 0 & a'_{2,1} & a'_{2,2} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_0 \\ b'_1 \\ b'_2 \end{pmatrix}$$

donde los valores de $a'_{i,j}$ los calculamos como:

$$a'_{i,j} = a_{i,j} - a_{i,k}\frac{a_{k,j}}{a_{k,k}}b_i = b_i - ai, k\frac{b_k}{a_{k,k}}$$

Si repetimos el procedimiento, ahora para x_1, tendremos un sistema dado como:

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ 0 & a'_{1,1} & a'_{1,2} \\ 0 & 0 & a''_{2,2} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_0 \\ b'_1 \\ b''_2 \end{pmatrix}$$

que corresponde a un sistema triangular superior que podemos solucionar utilizando sustitución hacia atrás.

Ejemplo.

Calcular el sistema triangular superior utilizando eliminación gaussiana.

$$5x + 2y + 1z = 3$$

$$2x + 3y - 3z = -10$$

$$1x - 3y + 2z = 4$$

Primer paso

$$5x + 2y + 1z = 3$$
$$0 + (11/5)y - (17/5)z = -(56/5)$$
$$0 - (17/5)y + (9/5)z = (17/5)$$

Segundo paso

$$5x + 2y + 1z = 3$$
$$0x + (11/5)y - (17/5)z = -(56/5)$$
$$0x - 0y - (38/11)z = -(153/11)$$

Ejemplo.

Resolver el sistema de ecuaciones

$$\begin{pmatrix} 3 & -1 & -1 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

Aplicando eliminación gaussiana nos queda.

$$\begin{pmatrix} 3 & -1 & -1 \\ 0 & 2/3 & -1/3 \\ 0 & 0 & 1/2 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 3/2 \end{pmatrix}$$

Ejemplo.

Un ejemplo de sistema donde es necesario hacer un cambio de renglón por renglón para que tenga solución es el siguiente sistema.

$$\begin{pmatrix} 1 & 2 & 6 \\ 4 & 8 & -1 \\ -2 & 3 & 5 \end{pmatrix}$$

Aplicando el primer paso de la eliminación gaussiana tenemos:

$$\begin{pmatrix} 1 & 2 & 6 \\ 0 & 0 & -25 \\ 0 & 7 & 17 \end{pmatrix}$$

note, que aparece un cero en el elemento 22, lo cual nos da un sistema sin solución. Permutando los renglones 2 y 3 el sistema tiene solución.

$$\begin{pmatrix} 1 & 2 & 6 \\ -2 & 3 & 5 \\ 4 & 8 & -1 \end{pmatrix}$$

La implementación de estos algoritmos en C es:

```
// operaciones_matrices.c

/**
 * <p>Title: Operaciones b·sicas con Matrices</p>
 *
 * <p>Description: Realiza las operaciones de suma, resta, multiplicaciÛn y
 * solucion de un sistema de equaciones</p>
 *
 * <p>Copyright: Copyright (c) 2005</p>
 *
 * <p>Company: UMSNH</p>
 *
 * @author Dr. FÈlix Calderon Solorio
 * @version 1.0
 */


#include <stdio.h>
#include <stdlib.h>

double **Matriz(int N, int M);
double *vector(int N);
double ** suma(double ** A, double ** B, int N, int M, int R, int S) ;
double ** resta(double ** A, double ** B, int N, int M, int R, int S) ;
double ** multiplica(double **A, double **B, int N, int M, int R, int S);
double * SolucionSL(double **a, double *b, int N);
void imprime_v(char texto[], double *fuente, int N) ;
void imprime_m(char texto[], double **fuente, int N, int M);


int main()
{
```

```
    double **A = Matriz(2,2);
    double **B = Matriz(2,2);

    A[0][0] = 1; A[0][1] = 2;
    A[1][0] = 3; A[1][1] = 4;

    imprime_m("A =", A, 2, 2);

    B[0][0] =  4; B[0][1] = 3;
    B[1][0] = -3; B[1][1] = 5;

    imprime_m("B =", B, 2, 2);

    double **C = suma(A, B, 2, 2, 2, 2);
    imprime_m("C = ", C, 2,2);

    double **D = resta(A, B, 2, 2, 2, 2);
    imprime_m("D = ",D, 2,2);

double **E = multiplica(A, B, 2, 2, 2, 2);
    imprime_m("E = ", E, 2,2);

double *Z = vector(2);
Z[0] = 10; Z[1] = 5;
double *x = SolucionSL(A, Z, 2);
    imprime_v("X =", x, 2);

return 1;

}


double **Matriz(int N, int M)
{
double **a;
        int i;

a = (double **) calloc (N, sizeof(double*));

for(i=0; i<N; i++)
a[i] = (double *) calloc(M, sizeof (double));
```

```
return a;
}

double *vector(int N)
{
double *a;

a = (double *) calloc (N, sizeof(double));

return a;
}

double ** suma(double ** A, double ** B, int N, int M, int R, int S)
{
  int i, j;

  if(N!=R || M!=S) return 0;

  double ** res = Matriz(N,M);

  for (i = 0; i < N; i++)
    for (j = 0; j < M; j++)
      res[i][j] = A[i][j] + B[i][j];

  return res;
}

double ** resta(double ** A, double ** B, int N, int M, int R, int S)
{

  int i, j;

  if(N!=R || M!=S) return 0;

  double ** res = Matriz(N,M);

  for (i = 0; i < N; i++)
    for (j = 0; j < M; j++)
      res[i][j] = A[i][j] - B[i][j];
```

```
  return res;
}

double ** multiplica(double **A, double **B, int N, int M, int R, int S)
{

  int i, j, k;

  if(M!=R) return 0;

  double **res = Matriz(N,S);

  for (i = 0; i < N; i++) {
    for (j = 0; j < S; j++) {
      res[i][j] = 0;

      for (k = 0; k < M; k++)
        res[i][j] += A[i][k] * B[k][j];
    }
  }
  return res;
}

double * SolucionSL(double **a, double *b, int N)
{
    int  i, j, k;
    double suma;

    double *x = vector(N);

    for (k = 0; k <= N - 2; k++) {
       for (i = k + 1; i <= (N - 1); i++) {
          b[i] -= a[i][k] * b[k] / a[k][k];
         for (j = N - 1; j >= k; j--)
                a[i][j] -= a[i][k] * a[k][j] / a[k][k];
        }
    }
    for (k = N - 1; k >= 0; k--) {
        suma = 0;
        for (j = k + 1; j < N; j++)
            suma += a[k][j] * x[j];
```

```
        x[k] = (b[k] - suma) / a[k][k];
    }
    return x;
}

void imprime_m(char texto[], double **fuente, int N, int M)
{
    printf("%s\n", texto);
    if(fuente == 0) return;
    int i, j;

    for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++) {
            printf("%f ", fuente[i][j]);
        }
    printf("\n");
    }
 }

void imprime_v(char texto[], double *fuente, int N)
{
   printf("%s \n", texto);
   if (fuente == 0) return;
   int i;

   for (i = 0; i < N; i++) {
     printf("%f \n", fuente[i]);
      }
}
```

6.10. Ejemplos

Búsquedas lineales

Secuencial

La forma más simple de búsqueda es la búsqueda secuencial. Esta búsqueda es aplicable a una tabla "organizada", ya sea como un arreglo o como una lista ligada. Supongamos que x es un arreglo de N llaves, de $x(0)$ a $x(N-1)$ y y el termino que deseamos encontrar en

el arreglo, el algoritmo de búsqueda queda como:

```
int busqueda_secuencial(double x[], double y, int N)
{
    int i;

    for(i=0; i<N; i++)
      if(x[i] == y) return i;

    return -1;
}
```

Búsqueda Binaria

El método de búsqueda más eficiente en una tabla secuencial sin usar índices o tablas auxiliares es el método de búsqueda binaria. Básicamente, se compara el argumento con la llave del elemento medio de la tabla. Sin son iguales, la búsqueda termina con éxito; en caso contrario, se busca de manera similar en la mitad superior o inferior de la tabla, según sea el caso.

```
int busqueda_binaria(double x[], double y, int N)
{
    int inicio = 0, fin = N, mitad;

    while(inicio <= fin)
    {
      mitad = (fin + inicio)/2;

      if(y == x[mitad]) return mitad;

      if(y < x[mitad]) fin = mitad -1;
      else inicio = mitad + 1;
    }
    return -1;
}
```

El siguiente código muestra la implementación de ambos y un programa principal para hacer un llamado a cada uno de los métodos de búsqueda antes mencionados.

```
/*
 * File:   busqueda_binaria.c
 * Author: felix
```

```
 *
 * Created on 16 de mayo de 2012, 12:28
 */

#include <stdio.h>
#include <stdlib.h>

int busqueda_secuencial(double x[], double y, int N)
{
    int i;

    for(i=0; i<N; i++)
      if(x[i] == y) return i;

    return -1;
}

int busqueda_binaria(double x[], double y, int N)
{
    int inicio = 0, fin = N, mitad;

    while(inicio <= fin)
    {
      mitad = (fin + inicio)/2;

      if(y == x[mitad]) return mitad;

      if(y < x[mitad]) fin = mitad -1;
      else inicio = mitad + 1;
    }
    return -1;
}

int main(int argc, char** argv) {
    double x[] = {1,2,4,7, 10, 11, 20, 21, 32};
    int N = 9, i;
    double y = 10;

    i = busqueda_secuencial(x, y, N);
    printf("El valor %f se encuentra en la posicion %d\n", y, i);
    printf("El valor %f se encuentra en la posicion %d\n", y,
```

```
      + busqueda_binaria(x, y, N));
    return (EXIT_SUCCESS);
}
```

Método de ordenamiento (burbuja)

El primer ordenamiento que presentamos es quizá el más ampliamente conocido entre los estudiantes que se inician en la programación. Una de las características de este ordenamiento es que es fácil de entender y programar. Aunque, es uno de los algoritmos mas ineficiente.

La idea básica subyacente en el ordenamiento de burbuja es pasar a través del arreglo de datos varias veces en forma secuencial. Cada paso consiste en la comparación de cada elemento en el arreglo con su sucesor (x[i] con x[i+1]) y el intercambio de los dos elementos si no están en el orden correcto. Considérese el siguiente arreglo de datos:

25 57 48 37 12 92 86 33

En el primer paso se realizan las siguientes operaciones:

x[0] con x[1] (25 con 57) no intercambio.

x[1] con x[2] (57 con 48) intercambio.

x[2] con x[3] (57 con 32) intercambio.

x[3] con x[4] (57 con 12) intercambio.

x[4] con x[5] (57 con 92) no intercambio.

x[5] con x[6] (92 con 86) intercambio.

x[6] con x[7] (92 con 33) intercambio.

Así después del primer paso, el arreglo está en el siguiente orden:

25 48 37 12 57 86 33 92

Observe que después del primer paso, el elemento mayor (en este caso 92) está en la posición correcta dentro del arreglo. En general x[n-i] estará en su posición correcta después de la iteración i. El método se lama ordenamiento de burbuja porque cada número “burbujea” con lentitud hacia su posición correcta. El conjunto completo de iteraciones es:

iteración 0: 25 57 48 37 12 92 86 33

iteración 1: 25 48 37 12 57 86 33 92

iteración 2: 25 37 12 48 57 33 86 92

iteración 3: 25 12 37 48 33 57 86 92

iteración 4: 12 25 37 33 48 57 89 92

iteración 5: 12 25 33 37 48 57 89 92

```
/*
 * File:   burbuja.c
 * Author: felix
 *
 * Created on 16 de mayo de 2012, 12:39
 */

#include <stdio.h>
#include <stdlib.h>

void burbuja(float *a, int N) {
    int i, j;
    float aux;

    do{
        j=0;
        for(i=0; i<N-1; i++) {
            if(a[i] > a[i+1]){
                aux    = a[i];
                a[i]   = a[i+1];
                a[i+1] = aux;
                j++;
            }
        }

    }while (j >0);
}

void imprime(float *a, int N) {
    int i;

    for(i=0; i<N; i++)
        printf("%f ", a[i]);
    printf("\n");
}
```

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

    float a[] = {1, 6, 4, 0, 3, -2};

    imprime(a, 6);
    burbuja(a, 6);
    imprime(a, 6);

    return (EXIT_SUCCESS);
}
```

Uso de cadenas de texto

7.1. Las cadenas como arreglos

Recordando la presentación de arreglos hecha en donde las cadenas están definidas como un arreglo de caracteres o un apuntador a una porción de memoria conteniendo caracteres ASCII. Una cadena en C es una secuencia de cero o más caracteres seguidas por un carácter NULL o

\0

:

F	A	C	U	L	T	A	D	\0

La cual se puede declarar como

```
char a[] = "FACULTAD"

char b[9] = "";

b[0] = 'F';
b[1] = 'A';
b[2] = 'C';
b[3] = 'U';
b[4] = 'L';
b[5] = 'T';
b[6] = 'A';
b[7] = 'D';
b[8] = '\0';
```

Es importante preservar el carácter de terminación NULL, ya que con éste es como C define y maneja las longitudes de las cadenas. Todas las funciones de la biblioteca estándar de C

lo requieren para una operación satisfactoria.

En general, aparte de algunas funciones restringidas en longitud (strncat(), strncmp() y strncpy()), al menos que se creen cadenas a mano, no se deberán encontrar problemas. Se deberán usar las funciones para manejo de cadenas y no tratar de manipular las cadenas en forma manual desmantelando y ensamblando cadenas.

7.2. Implementación de algunas funciones para manejar cadenas

En esta sección se muestran algunas funciones implementadas para ver el manejo de cadenas de caracteres y en la siguiente sección se muestra algunas funciones standard.

7.2.1. Impresión de Cadenas como arreglos

El siguiente es un ejemplo de como imprimir una cadena de caracteres con la función printf. Adicionalmente muestra el manejo de arreglos bidimensionales de caracteres.

```
/*
 * File:   arreglos_cadenas.c
 * Author: felix
 *
 * Created on 3 de noviembre de 2011, 13:30
 */

#include <stdio.h>
#include <stdlib.h>


void ejemplo_01() {
    char a[3][5] = {"Uno", "dos", "tres"};

    int i;

    for(i=0; i<3; i++)
        printf("%s\n", a[i]);
}

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

```
    int i;

    if(argc > 1)
        for(i=0; i<argc; i++)
            printf("%d %s\n", i, argv[i]);

    return (EXIT_SUCCESS);
}
```

Note que los argumentos de main contienen información del número de parámetros argc y las cadenas de caracteres dados en argv.

7.2.2. Manejo de cadenas de caracteres con apuntadores

Dado que las cadenas de caracteres son arreglos, estos pueden manejarse por apuntadores. El siguiente ejemplo muestra como se pueden almacenar las cadenas en un arreglo y en un apuntador.

```
// Ejemplo ec082.cpp

/**
*  Este programa muestra los diferentes formatos para i
*  imprimir caracteres y cadenas de caracteres
*/
#include <stdio.h>

int main()
{
char a = 'A';
char cadena[] = "Esto es una cadena de texto";
char *cadenap = "Esto tambien es una cadena de texto";

printf("%c\n", a);
printf("%s\n", "Esto es una cadena de texto");
printf("%s\n", cadena);
printf("%s\n", cadenap);

return 0;
}
```

7.2.3. Longitud de una cadena de Texto

Dado que una cadena de texto tiene un terminador de cadena dado por '
0', podemos hacer un ciclo para contar los caracteres de una cadena. Note que la condición de terminación es que el i-esimo carácter sea '
0' tal como se muestra en el siguiente código.

```
int longitud(char *a) {
    int i, c=0;
    for(i=0; a[i]!= '\0'; i++)
        c++;
    return c;
}
```

ver cadenas_texto.c

7.3. Uso de las funciones proporcionadas por las librerías

Todas las funciones para manejo de cadenas tienen su prototipo en:

```
#include <string.h>
```

Las funciones m·s comunes son descritas a continuación:

char *strcpy(const char *dest, const char *orig) – Copia la cadena de caracteres apuntada por orig (incluyendo el car·cter terminador

```
'\0'
```

) al vector apuntado por dest. Las cadenas no deben solaparse, y la de destino, debe ser suficientemente grande como para alojar la copia.

int strcmp(const char *s1, const char *s2) – Compara las dos cadenas de caracteres s1 y s2. Devuelve un entero menor, igual o mayor que cero si se encuentra que zs1 es, respectivamente, menor que, igual a, o mayor que s2.

char *strerror(int errnum) – Devuelve un mensaje de error que corresponde a un número de error.

int strlen(const char *s) – Calcula la longitud de la cadena de caracteres.

char *strncat(char *s1, const char *s2, size_t n) – Agrega n caracteres de s2 a s1.

int strncmp(const char *s1, char *s2, size_t n) – Compara los primeros n caracteres de dos cadenas.

char *strncpy(const char *s1, const char *s2, size_t n) – Copia los primeros n caracteres de s2 a s1.

strcasecmp(const char *s1, const char *s2) – versión que ignora si son mayúsculas o minúsculas de strcmp().

strncasecmp(const char *s1, const char *s2, size_t n) – versiÛn insensible a mayúsculas o minúsculas de strncmp() que compara los primeros n caracteres de s1.

El uso de muchas funciones es directo, por ejemplo:

```
char *s1 = "Hola"; char *s2; int longitud;

longitud = strlen("Hola");  /* long = 4 */ (void) strcpy(s2,s1);
```

Observar que tanto strcat() y strcopy() regresan una copia de su primer argumento, el cual es el arreglo destino. Observar también que orden de los argumentos es arreglo destino seguido por arreglo fuente lo cual a veces es una situación para hacerlo incorrectamente.

La función strcmp() compara lexicográficamente las dos cadenas y regresa:

Menor que cero – si s1 es léxicamente menor que s2; Cero – si s1 y s2 son léxicamente iguales; Mayor que cero – si s1 es léxicamente mayor que s2; Las funciones de copiado strncat(), strncmp() y strncpy() son versiones m·s restringidas que sus contrapartes más generales. Realizan una tarea similar, pero solamente para los primeros n caracteres. Observar que el carácter de terminación NULL podría ser violado cuando se usa estas funciones, por ejemplo:

```
char *s1 = "Hola"; char *s2 = 2; int longitud = 2;

(void) strncpy(s2, s1, longitud); /* s2 = "Ho" */
```

donde s2 no tiene el terminador NULL.

Búsqueda en cadenas

La biblioteca también proporciona varias funciones de búsqueda en cadenas

char *strchr(const char *s, int c) – Devuelve un puntero a la primera ocurrencia del carácter c en la cadena de caracteres s.

char *strrchr(const char *s, int c) – Encuentra la última ocurrencia del caracter c en la cadena.

char *strstr(const char *s1, const char *s2) –Localiza la primera ocurrencia de la cadena s2 en la cadena s1.

char *strpbrk(const char *s1, const char *s2)– Regresa un apuntador a la primera ocurrencia en la cadena s1 de cualquier carácter de la cadena s2, o un apuntador nulo si no hay un carácter de s2 que exista en s1.

size_t strspn(const char *s1, const char *s2)– Calcula la longitud del segmento inicial de s1 que consta únicamente de caracteres en s2.

size_t strcspn(const char *s1, const char *s2) – Regresa el número de caracteres al principio de s1 que no coinciden con s2.

char *strtok(char *s1, const char *s2) – Divide la cadena apuntada a s1 en una secuencia de "tokens", cada uno de ellos esta delimitado por uno o m·s caracteres de la cadena apuntada por s2.

Las funciones strchr() y strrchr() son las más simples de usar, por ejemplo:

```
char *s1 = "Hola";
char *resp;

resp = strchr(s1,'l');
```

Después de la ejecución, resp apunta a la localidad s1 + 2.

La función strpbrk() es una función más general que busca la primera ocurrencia de cualquier grupo de caracteres, por ejemplo:

```
char *s1 = "Hola"; char *resp;

res = strpbrk(s1,"aeiou");
```

En este caso, resp apunta a la localidad s1 + 1, la localidad de la primera o.

La función strstr() regresa un apuntador a la cadena de búsqueda especificada o un apuntador nulo si la cadena no es encontrada. Si s2 apunta a una cadena de longitud cero (esto es, la cadena ""), la función regresa s1. Por ejemplo:

```
char *s1 = "Hola"; char *resp;

resp = strstr(s1,"la");
```

la cual tendrá resp = s1 + 2.

La función strtok() es un poco m·s complicada en cuanto a operación. Si el primer argumento no es NULL entonces la función encuentra la posición de cualquiera de los caracteres del segundo argumento. Sin embargo, la posición es recordada y cualquier llamada subsecuente a strtok() iniciará en esa posición si en estas subsecuentes llamadas el primer argumento no es NULL. Por ejemplo, si deseamos dividir la cadena s1 usando cada espacio e imprimir cada "token" en una nueva línea haríamos lo siguiente:

```
char s1[] = "Hola muchacho grande"; char *t1;

for ( t1 = strtok(s1," ");
      t1 != NULL;
      t1 = strtok(NULL, " ") )
    printf("%s\n",t1);
```

Se emplea un ciclo for en una forma no regular de conteo:

En la inicialización se llama a la función strtok() con la cadena s1. Se termina cuando t1 es NULL. Se esta asignando tokens de s1 a t1 hasta la terminación llamando a strtok() con el primer argumento NULL.

Prueba de clasificación de caracteres: <ctype.h>

El argumento de las funciones siguientes es un int cuyo valor debe ser EOF o representable por un unsigned char. El valor de retorno es un int. Las funciones regresan cero si el argumento satisface la condición y un valor diferente de cero si no lo hace.

FunciÛn	DescripciÛn
islower(c)	letra minúscula
isupper(c)	letra mayúscula
isalpha(c)	letra mayúscula o minúscula
isdigit(c)	dígito decimal
isalnum(c)	letra mayúscula o minúscula o dígito decimal
iscntrl(c)	carácter de control
isgraph(c)	carácter de impresión excepto espacio
isprint(c)	carácter de impresión incluyendo espacio
ispunct(c)	carácter de impresión excepto espacio, letra o dígito
isspace(c)	espacio, avance de línea, nueva línea, retorno de carro, tabulador, tabulador vertical
isxdigit(c)	dígito hexadecimal

Funciones de conversión:

int tolower(int c) convierte c a letra minúscula

int toupper(int c) convierte c a letra mayúscula

Conversión de cadenas

Existen unas cuantas funciones para convertir cadenas a enteros, enteros largos y valores flotantes, las cuales se encuentran en la librerÌa <stdlib.h>. Estas son:

double atof(const char *cadena) : Convierte una cadena a un valor flotante.

int atoi(const char *cadena) : Convierte una cadena a un valor entero.

int atol(const char *cadena) : Convierte una cadena a un valor entero largo.

double strtod(const char *cadena, char **finap) : Convierte una cadena a un valor de punto flotante.

double strtol(const char *cadena, char *finap, int base) : Convierte una cadena a un entero largo de acuerdo a una base dada, la cual deber· estar entre 2 y 36 inclusive.

unsigned long strtoul(const char *cadena, char *finap, int base) : Convierte una cadena a un entero largo sin signo.

Varias de las funciones se pueden usar en forma directa, por ejemplo:

```
char *cad1 = "100";
char *cad2 = "55.444";
char *cad3 = "1234";
char *cad4 = "123cuatro";
char *cad5 = "invalido123";
char *cad6 = "123E23Hola";
char *cad7;

int i; float f:

i = atoi(cad1);    /* i = 100 */
f = atof(cad2);    /* f = 55.44 */
i = atoi(cad3);    /* i =1234*/
i = atoi(cad4);    /* i = 123 */
i = atoi(cad5);    /* i =0 */
f = strtod(cad6, &cad7);  /* f=1.230000E+25 y cad7=hola*/
```

Nota:

Los caracteres en blanco son saltados. Caracteres ilegales son ignorados. Si la conversión no puede ser hecha se regresa cero y errno es puesta con el valor ERANGE.

Estructuras, Uniones y enumeraciones

8.1. Estructuras en C

Una Estructura es una colección de variables simples, que pueden contener diferentes tipos de datos. Es un tipo de dato definido por el usuario. Son también conocidas como Registros. Ayudan a organizar y manejar datos complicados en programas debido a que agrupan diferentes tipos de datos a las que se les trata como una sola unidad en lugar de ser vistas como unidades separadas.

La Declaración de estructuras en programas C es un nuevo tipo de datos denominado tipo Estructura y declarar una variable de este tipo.

En la definición del tipo de estructura, se especifican los elementos que la componen asÌ como sus tipos. Cada elemento es llamado miembro (similar a un campo de un registro).

```
struct tipo_estructura { Declaración de los miembros };
```

Después de definir un tipo estructura, se puede declarar una o más variables de ese tipo de la siguiente forma:

```
struct tipo_estructura [variables];
```

Ejemplo

```
struct articulo {
  int codigo;
  char nombre[20];
  float peso, longitud;
};
```

Definición de variables de esta estructura : struct a, b; ó también se pueden definir la estructura y sus variables al mismo tiempo:

```
struct articulo {
  int codigo;
```

```
  char nombre[20];
  float peso, longitud;
} a, b;
```

El Acceso a los Miembros de la Estructura es por medio del operador Punto de la siguiente manera:

```
a.codigo = 1234;
b.nombre = "escoba";
b.peso = 23;
```

Los miembros de las Estructuras también se pueden inicializar individualmente o bien, todos simultáneamente. La inicialización individual es por medio del operador punto, para la estructura completa es con caracteres llaves y los valores de los miembros separados por coma. Iniciando solo por miembros: a.codigo=1234;

Iniciando Estructura completa:

```
articulo a={1234, "escoba", 23, 1.50};
```

8.1.1. Ejemplo de manejo de estructuras

```
// Ejemplo ej045.c

/**
 * Title: Manejo de estructuras en C
 * Description: Crea una estructura con datos
 * desde el teclado
 * Copyright: Copyright (c) 2005
 * Company: UMSNH
 * author Dr. Felix Calderon Solorio
 * version 1.0
 */

#include <stdio.h>
#include <conio.h>

struct datos {
    char no_control[9];
    char nombre[30];
    int edad;
    char sexo;
```

```
    char domicilio[50];
};


void imprime(struct datos a);

int main() {
    struct datos alumno;

    printf("\nIntroduce los datos del alumno");
    printf("\nNo Control : ");
    scanf("%s", alumno.no_control);
    printf("\nNombre     : ");
    scanf("%s", alumno.nombre);
    printf("\nEdad       : ");
    scanf("%d", &alumno.edad);
    printf("\nSexo       : ");
    alumno.sexo = getchar();
    alumno.sexo = getchar();
    printf("\nDireccion  : ");
    scanf("%s", alumno.domicilio);

    imprime(alumno);

    return 1;
}

void imprime(struct datos a) {
    printf("No de control %s\n", a.no_control);
    printf("Nombre %s\n", a.nombre);
    printf("Edad %d\n", a.edad);
    printf("Sexo %c\n", a.sexo);
    printf("Direccion %s\n", a.domicilio);
}
```

8.1.2. Arreglos de Estructuras

Arreglos de Estructuras pueden ser construidas, es decir, conceptuar a los elementos de un arreglo como estructuras, esto se puede hacer de la siguiente manera:

```
struct datos {
```

```
    char no_control[9];
    char nombre[30];
    int edad;
    char sexo;
    char domicilio[50];
};

struct datos alumnos[35];
```

Para tener acceso a una determinada estructura , se indexa el nombre de la estructura de la siguiente manera: alumnos[i].nombre; aquí sabemos el nombre del alumno que se encuentra en la posición i del arreglo, y de esta manera para todos los datos. Si se desea inicializar los valores, se podría hacer de la siguiente manera:

```
struct datos.alumno[5]={96147523, "Luis", 22, "m", "pino 45"};
```

8.1.3. Estructuras Anidadas

El siguiente ejemplo muestra una estructura anidada. Las estructuras que hemos visto han sido muy sencillas aunque útiles. Es posible definir estructuras conteniendo docenas y aún cientos ó miles de elementos pero sería ventajoso para el programador no definir todos los elementos en una pasada sino utilizar una definición de estructura jerárquica.

```
// Ejemplo ec046.cpp

/**
 * Title: Manejo de estructuras en C
 * Description: Crea una estructura con datos
 * desde el teclado
 * Copyright: Copyright (c) 2005
 * Company: UMSNH
 * author Dr. Felix Calderon Solorio
 * version 1.0
 */

#include <string.h> #include <stdio.h>

struct persona {
    char nombre[25];
    int edad;
```

```
    char estado; // C = casado, S = soltero
};

struct datos {
    int calificacion;
    struct persona descripcion;
    char comida[25];
};

void imprime(struct datos a);

int main() {
    struct datos estudiante[53];
    struct datos maestro, sub;

    maestro.calificacion = 94;
    maestro.descripcion.estado = 'C';
    strcpy(maestro.descripcion.nombre, "Lisseth Gil");
    strcpy(maestro.comida, "Chocolate de Ron");

    sub.descripcion.edad = 87;
    sub.descripcion.estado = 'C';
    strcpy(sub.descripcion.nombre, "Abuela Pata");
    sub.calificacion  = 73;
    strcpy(sub.comida, "Maiz y agua");

    estudiante[1].descripcion.edad = 15;
    estudiante[1].descripcion.estado = 'S';
    strcpy(estudiante[1].descripcion.nombre, "Bill, Griton");
    strcpy(estudiante[1].comida, "Crema de cacahuate");
    estudiante[1].calificacion = 77;

    estudiante[7].descripcion.edad = 14;
    estudiante[12].calificacion = 87;

    imprime(sub);

    return 1;
}
```

```
void imprime(struct datos a) {
    printf("Nombre %s\n", a.descripcion.nombre);
    printf("Edad %d\n", a.descripcion.edad);
    printf("Estado %c\n", a.descripcion.estado);
    printf("calificacion %d\n", a.calificacion);
}
```

La primera estructura contiene tres elementos pero no le sigue ninguna variable definida, sólo una estructura, pero como incluimos un nombre al principio de la estructura, la estructura es llamada persona. La palabra persona puede utilizarse para referirse a la estructura pero no a cualquier variable de éste tipo de estructura, se trata por lo tanto de un nuevo tipo que hemos definido y lo podemos utilizar de la misma manera en que usamos un int, char o cualquier otro tipo que existe en C. La única restricción es que éste nuevo tipo debe estar siempre asociado con la palabra clave struct.

La siguiente definición de estructura contiene tres campos siendo el segundo la estructura previamente definida la cual llamamos persona. La variable de tipo persona se llama descripción, así la nueva estructura contiene dos variables simples, calificación y una cadena llamada comida, y la estructura llamada descripción. Como descripción contiene tres variables, la nueva estructura tiene entonces cinco variables, a ésta estructura le hemos dado el nombre de datos, lo cual es otro tipo definido. Finalmente, dentro de la función main ( ) definimos un array de 53 variables cada una con la estructura definida por el tipo datos, y cada una con el nombre estudiante, en total hemos definido 53 veces 5 variables, cada una de las cuales es capaz de almacenar datos. Como tenemos la definición de un nuevo tipo podemos utilizarla para a su vez definir dos variables. Las variables maestro y sub están definidas en la línea 20 como variables de tipo datos por lo que cada una de éstas dos variables contienen 5 campos en los cuales podemos almacenar datos.

En las líneas 22 a 26 del programa asignamos valores a cada uno de los campos de maestro. El primero es el campo calificación y es manejado como las otras estructuras que hemos estudiado porque no forma parte de la estructura anidada. Enseguida deseamos asignar un valor a edad el cual es parte de la estructura anidada. Para acceder a éste campo empezamos con el nombre de la variable maestro al cual le concatenamos el nombre del grupo descripción, y entonces debemos definir en cuál campo de la estructura anidada estamos interesados por lo que concatenamos el nombre de la variable edad. El estado de los maestros se manejan de la misma manera que su edad pero los últimos dos campos son cadenas asignadas utilizando la función strcpy ( ). Observe que los nombres de las variables en la función strcpy ( ) se consideran como una unidad aunque estén compuestas de varias partes.

Compile y ejecute el programa, probablemente obtenga un aviso sea de error ó advertencia respecto a un desbordamiento de memoria.

Lo que esto significa es que el programa requiere más memoria que la asignada por el compilador por lo que es necesario incrementar el tamaño de stacks, el método para hacer esto varía de un compilador a otro.

8.1.4. Manejo de estructuras con apuntadores

Cuando se manejan estructuras como apuntadores el acceso a los campos es a través de una flecha. Por ejemplo si definimos una estructura como

```
struct per {
    char *nombre;
    char *domicilio;
    int edad;
    char sexo;
};
```

y defininos

```
struct per * a
```

cada uno de los campo se accesa como a-¿nombre, a-¿domicilio, a-¿edad y a-¿sexo.

En general es mucho mas robusto el manejo de las estructuras para generar arreglos. El siguiente es un ejemplo de como se debe hacer este manejo.

```
/*
 * File:   persona.c
 * Author: felix
 *
 * Created on 23 de noviembre de 2011, 11:59
 */

#include <stdio.h>
#include <stdlib.h>

/*
 * Ejemplo de manejo de estructuras.
 * Se introduce los datos refetentes a personas.
 */

typedef struct per * persona;

struct per {
```

```
    char *nombre;
    char *domicilio;
    int edad;
    char sexo;
};

int longitud(char *a) {
    int i;
    for(i=0; a[i]!=0; i++);
    return i;
}

persona nueva(char *n, char *d, int e, char s) {
    persona aux;
    int i, nc;

    aux = (persona) calloc(1, sizeof(struct per));

    nc = longitud(n);
    aux->nombre = (char *) calloc(nc, sizeof(char));
    for(i=0; i<nc; i++)
        aux->nombre[i] = n[i];

    nc = longitud(d);
    aux->domicilio = (char *) calloc(nc, sizeof(char));
    for(i=0; i<nc; i++)
        aux->domicilio[i] = d[i];

    aux->edad = e;
    aux->sexo = s;

    return aux;
}


void imprime(persona a) {
    printf("Nombre    = %s\n", a->nombre);
    printf("Domicilio = %s\n", a->domicilio);
    printf("Edad      = %d\n", a->edad);
    printf("sexo      = %c\n", a->sexo);
}
```

```
int main(int argc, char** argv) {
    persona a;
    persona *b;
    int i;
    a = nueva("Juan", "El retajo", 10, 'h');
    imprime(a);

    b = (persona *) calloc(3, sizeof(persona));
    b[0] = nueva("Carlos", "CU", 20, 'h');
    b[1] = nueva("Ana", "cerca de CU", 21, 'm');
    b[2] = nueva("Luis", "no se ", 40, 'h');

    for(i=0; i<3; i++)
        imprime(b[i]);

    return (EXIT_SUCCESS);
}
```

8.1.5. Manejo de Fracciones

El siguiente es un ejemplo de manejo de fracciones utilizando estructuras de datos. El código tiene implementado funciones para suma, resta, multiplicación y división de fracciones, adicionalmente tiene el algoritmo de Euclides para simplificar fracciones y una función para imprimir fracciones

```
#include <stdio.h>
#include <stdlib.h>

typedef struct razon * fraccion;

int EuclidesMCD(int a, int b);
int EuclidesMCM(int a, int b);
fraccion nueva(int n, int d);
void imprime(fraccion a);
fraccion suma(fraccion a, fraccion b);
fraccion resta(fraccion a, fraccion b);

struct razon {
   int numerador;
```

```
   int denominador;
};

int EuclidesMCD(int a, int b){

    int aux;
    int mayor = a > b ? a : b;
    int menor = a < b ? a : b;

    do{
        aux = menor;
        menor = mayor % menor;
        mayor = aux;
    } while(menor!=0);
    return mayor;
}

int EuclidesMCM(int a, int b) {
    return(a/EuclidesMCD(a,b))*b;
}

fraccion nueva(int n, int d) {
   fraccion aux = (fraccion) calloc(1, sizeof(struct razon));
   int x;
   x = EuclidesMCD(n, d);
   aux -> numerador = n/x;
   aux -> denominador = d/x;
   return aux;
}

void imprime(fraccion a) {
   printf("%d/%d \n", a->numerador, a->denominador);
}

fraccion suma(fraccion a, fraccion b) {
   fraccion aux;

   int num, den;
   den = a->denominador * b->denominador;
   num = a->numerador * b->denominador + a->denominador * b->numerador;
```

```
   aux = nueva(num, den);

   return aux;
}

fraccion resta(fraccion a, fraccion b) {
   fraccion aux;

   int num, den;
   den = a->denominador * b->denominador;
   num = a->numerador * b->denominador - a->denominador * b->numerador;

   aux = nueva(num, den);

   return aux;
}

fraccion multiplicacion(fraccion a, fraccion b){

   return nueva(a->numerador*b->numerador,  a->denominador*b->denominador);
}

fraccion division(fraccion a, fraccion b){

   return nueva(a->numerador*b->denominador,  a->denominador*b->numerador);
}

int main(){
   fraccion a = nueva(20, 30);
   fraccion b = nueva(4, 20);
   fraccion c;

   imprime(a);
   imprime(b);
   c = suma(a, b);
   imprime(c);
   c = resta(a, b);
   imprime(c);
   c = multiplicacion(a, b);
   imprime(c);
   c = division(a, b);
```

```
   imprime(c);
}
```

8.1.6. Estructuras a nivel de bits

```
/*
 * File:   cartas.c
 * Author: felix
 *
 * Created on 1 de diciembre de 2011, 17:59
 */

#include <stdio.h>
#include <stdlib.h>

/*
 *
 */

struct cartaBit {
   unsigned cara : 4;  /* 4 bits; 0-15 */
   unsigned palo : 2;  /* 2 bits; 0-3  */
   unsigned color : 1; /* 1 bit; 0-1   */
}; /* fin de la estructura cartaBit    */

typedef struct cartaBit Carta; /* nuevo nombre de tipo para la estructura cartaBit */
void llenaMazo( Carta * const wMazo );      /* prototipo */
void repartir( const Carta * const wMazo ); /* prototipo */

int main(){
   Carta mazo[ 52 ]; /* crea el arreglo de Cartas */

   llenaMazo( mazo );
   repartir( mazo );
   return 0; /* indica terminacion exitosa */

}

/* fin de main */
/* inicializa Carta */
```

```
void llenaMazo( Carta * const wMazo )
{
   int i; /* contador */
    /* ciclo a traves de wMazo */
   for ( i = 0; i <= 51; i++ ) {
      wMazo[ i ].cara = i % 13;
      wMazo[ i ].palo = i / 13;
      wMazo[ i ].color = i / 26;   } /* fin de for */

} /* fin de la funcion llenaMazo */

/* muestra las barajas con formatoc de dos columnas; el subindice de las
 * barajas de 0 a 25 es k1 (columna 1); el subindice de las barajas de 26
 * a 51 es k2 (columna 2) */

void repartir( const Carta * const wMazo )
{
   int k1; /* subindice 0-25 */
   int k2; /* subindice 26-51 */
   /* ciclo a traves de wMazo */
   for ( k1 = 0, k2 = k1 + 26; k1 <= 25; k1++, k2++ ) {
      printf( "Carta:%3d  Palo:%2d  Color:%2d   \n",
         wMazo[ k1 ].cara, wMazo[ k1 ].palo, wMazo[ k1 ].color );
      printf( "Carta:%3d  Palo:%2d  Color:%2d \n",
         wMazo[ k2 ].cara, wMazo[ k2 ].palo, wMazo[ k2 ].color );
   } /* fin de for */

} /* fin de la funcion repartir */
```

8.2. Uniones en C

Uniones. Las uniones son similares a las estructuras, con la diferencia de que en las uniones se almacenan en los campos solapándose unos con otros en la misma disposición; al contrario que en las estructuras, al contrario que en las estructuras, donde los campos se almacenan unos a continuación de otros. En esencia, las uniones sirven para ahorrar espacio en memoria. Para Almacenar los miembros de una unión, se requiere una zona de memoria igual a la que ocupa el miembro mas largo de la unión. Todos los miembros son almacenados en el mismo espacio de memoria y comienzan en la misma dirección. El valor

almacenado es sobre escrito cada vez que se asigna un valor al mismo miembro o a un miembro diferente, aquí radica la diferencia con las estructuras.

La Declaración de Uniones es similar a la de las estructuras:

```
union tipo_union {
 declaracion_de_miembros;
 } ;

 tipo_union
```

Después de definir un tipo union, se puede declarar una o mas variables de ese tipo de la siguiente forma:

```
union tipo_union [variables];
```

Una declaración union define un tipo. La llave derecha que termina la lista de miembros puede ser seguida por una lista de variables de la siguiente manera:

```
union tipo_union{ ... } variable;
```

Ejemplo

```
union datos {
  char a;
  int b;
  float c
};

union datos p;
```

Dicho de una forma simple, una unión le permite manejar los mismos datos con diferentes tipos, ó utilizar el mismo dato con diferente nombre, a continuación le presento un ejemplo:

```
// union01.c

/**
 * Title: Manejo de uniones en C
 * Description: Crea una union de datos
 * desde el teclado
 * Copyright: Copyright (c) 2005
 * Company: UMSNH
 * author Dr. Felix Calderon Solorio
 * version 1.0
 */
```

```
#include <stdio.h>

union
{
int valor;

struct
{
char primero;
char segundo;
} mitad;
} numero;


int main()
{
long indice = 34000L;

numero.valor = indice;

printf("%4x %2x %2x\n", numero.valor, numero.mitad.primero, numero.mitad.segundo);

return 1;

}
```

La ejecución muestra los siguientes resultados.

```
    84d0 ffffffd0 ffffff84
```

ver [**?**]

En éste ejemplo tenemos dos elementos en la unión, la primera parte es el entero llamado valor el cual es almacenado en algún lugar de la memoria de la computadora como una variable de dos bytes. El segundo elemento está compuesto de dos variables de tipo char llamadas primero y segundo. Estas dos variables son almacenadas en la misma ubicación de almacenamiento que valor porque esto es precisamente lo que una unión hace, le permite almacenar diferentes tipos de datos en la misma ubicación física. En éste caso Usted puede poner un valor de tipo entero en valor y después recobrarlo en dos partes utilizando primero y segundo, ésta técnica es utilizada a menudo para empaquetar bytes cuando, por ejemplo, combine bytes para utilizarlos en los registros del microprocesador. La unión no es utilizada frecuentemente y casi nunca por programadores principiantes, en este momento no necesita

profundizar en el empleo de la unión así que no dedique mucho tiempo a su estudio, sin embargo no tome a la ligera el concepto de la unión, podría utilizarlo a menudo.

El siguiente es un ejemplo donde de manera indistinta una variable puede ser declarada como entero o doble. Esto no debe crear confusión ya que esto permite tener datos de tipo genérico donde con un nombre de variable lo vemos de una manera y con otra variable de otra forma.

```
/*
 * File:   ejemplo_union.c
 * Author: felix
 *
 * Created on 1 de diciembre de 2011, 20:04
 */

#include <stdio.h>
#include <stdlib.h>

/* definicion de la union numero */

union numero {
   int x;
   double y;
}; /* fin de la union numero */

int main()
{   union numero valor; /* define la variable de union */

   valor.x = 100; /* coloca un entero dentro de la union */
   printf( "%s\n%s\n%s%d\n%s%f\n",
      "Coloca un valor en el miembro entero",
       "e imprime ambos miembros.",
      "int:   ", valor.x,
      "double: ", valor.y );

   valor.y = 100.0; /* coloca un double dentro de la misma union */
   printf( "%s\n%s\n%s%d\n%s%f\n",
      "Coloca un valor en el miembro flotante",
      "e imprime ambos miembros.",
      "int:   ", valor.x,
      "double: ", valor.y );
   return 0; /* indica terminacion exitosa */
```

```
} /* fin de main */
```

8.2.1. Ejemplo Tipo Genéricos

El siguiente es un ejemplo donde manejamos un tipo de dato genérico. Este tipo tiene la posibilidad de ser al mismo tiempo un carácter, un entero, un flotante o una cadena de caracteres. Un identificador permite diferenciar cada tipo al momento de imprimir y todos los datos son la unión de los tipos mencionados.

```
/*
 * File:   tipo_generico.c
 * Author: felix
 *
 * Created on 25 de junio de 2012, 10:17
 */


#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct gen * generico;

#define CHAR   0
#define INT    1
#define FLOAT  2
#define STRING 3


struct gen {
  int tipo;  // tipo de datos
  union {
    char cval;
    int ival;
    float fval;
    char *sval;
  } valor;  // valores
};

generico nueva_entero(int a){
```

```
  generico aux;
  aux  = (generico) calloc(1, sizeof(struct gen));
  aux->tipo = INT;
  aux->valor.ival = a;
  return aux;
}

generico nueva_char(char a){
  generico aux;
  aux  = (generico) calloc(1, sizeof(struct gen));
  aux->tipo = CHAR;
  aux->valor.cval = a;
  return aux;
}

generico nueva_flotante(float a){
  generico aux;
  aux  = (generico) calloc(1, sizeof(struct gen));
  aux->tipo = FLOAT;
  aux->valor.fval = a;
  return aux;
}

generico nueva_cadena(char *a){
  generico aux;
  int N;
  aux  = (generico) calloc(1, sizeof(struct gen));
  aux->tipo = STRING;
  N = strlen(a);
  aux->valor.sval = (char *) calloc(N, sizeof(char));
  strcpy(aux->valor.sval,a );
  return aux;
}

void imprime(generico a){
  if(a->tipo == CHAR) printf("%c\n", a->valor.cval);
  else if(a->tipo == INT) printf("%d\n", a->valor.ival);
  else if(a->tipo == FLOAT) printf("%f\n", a->valor.fval);
  else if(a->tipo == STRING) printf("%s\n", a->valor.sval);
  else printf("Tipo no valido\n");
}
```

```
generico * vector(int N){
  return (generico *) calloc(N, sizeof(generico));
}

int main(int argc, char** argv) {
    generico *a;
    a = vector(3);
    int i;
    a[0] = nueva_entero(2);
    a[1] = nueva_char('c');
    a[2] = nueva_flotante(15.4);

    for(i=0; i<3; i++)
        imprime(a[i]);

    generico b = nueva_cadena("Hola Mundo");

    imprime(b);
    return (EXIT_SUCCESS);
}
```

Definición del proyecto de aplicación

9.1. Proyecto

Una técnica utilizada para representar imágenes y/o fotos consiste en sustituir el color o tono de gris por un carácter ASCII. Así una imagen de presidente Obama lucirá como

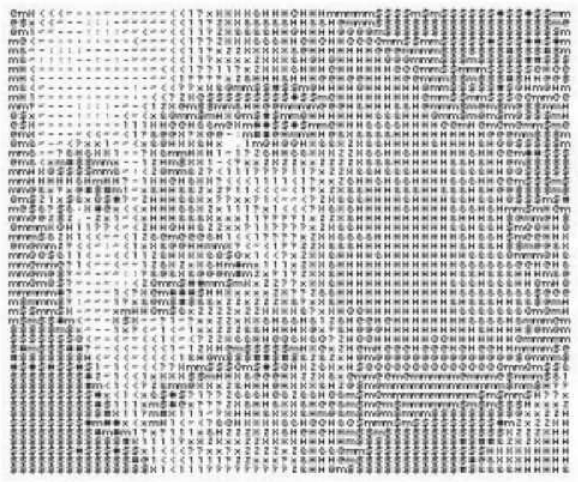


Figura 9.2: Ejemplo de imagen generada

El proyecto consistirá en leer la imagen en un formato png, procesar los datos de la imagen y en un archivo de texto dejar la imagen creada con caracteres.

9.2. Proyecto

Una imagen fotográfica por si misma es un trabajo artístico, sin embargo no siempre las personas están satisfechas con la foto obtenida y desean agregar algún efecto especial a sus

fotografías. Este proyecto intenta crear un efecto en la imágenes, de tal suerte que estas luzcan como una imagen hecha a mano y pintada con acuarelas. Para ello se calculan los bordes de la imagen y se suman a la imagen original pesando de manera diferente cada una de la imágenes. En la figura 9.3 se presenta el resultado que deseamos, del lado izquierdo en la figura 9.3(a) la imagen original y del lado derecho en la figura 9.3(b) el resultado que deseamos.

(a) ***Imagen Original*** **(b)** ***Imagen modificada***

Figura 9.3: Efecto aplicado sobre una imagen

Lectura y/o Escritura de archivos

10.1. Introducción a los archivos

Los archivos son documentos donde almacenamos y leemos datos. Sería muy tedioso tener que escribir mas de 100 datos de entrada para un programa, procesarlos y copiar de pantalla los resultados. En estos casos resulta mas obvio leer un archivo, procesar la información y guardar en un archivos los datos de salida.

El procedimiento de lectura y/o escritura lleva tres pasos

1. Abrir el archivo de trabajo,
2. Leer o guardar la información y
3. Cerrar el archivo

10.2. Abrir un archivo

Tipo FILE

C define la estructura de datos FILE en el fichero de cabecera "stdio.h" para el manejo de ficheros. Nosotros siempre usaremos punteros a estas estructuras.

La definición de esta estructura depende del compilador, pero en general mantienen un campo con la posición actual de lectura/escritura, un buffer para mejorar las prestaciones de acceso al fichero y algunos campos para uso interno.

Función fopen:

Sintaxis:

FILE *fopen(char *nombre, char *modo); Esta función sirve para abrir y crear ficheros en disco. El valor de retorno es un puntero a una estructura FILE. Los parámetros de entrada son:

nombre: una cadena que contiene un nombre de fichero valido, esto depende del sistema operativo que estemos usando. El nombre puede incluir el camino completo.

modo: especifica en tipo de fichero que se abrirá o se creará y el tipo de datos que puede contener, de texto o binarios:

r: sólo lectura. El fichero debe existir.

w: se abre para escritura, se crea un fichero nuevo o se sobreescribe si ya existe.

a: añadir, se abre para escritura, el cursor se sitúa al final del fichero. Si el fichero no existe, se crea.

r+: lectura y escritura. El fichero debe existir.

w+: lectura y escritura, se crea un fichero nuevo o se sobre escribe si ya existe. a+: añadir, lectura y escritura, el cursor se sitúa al final del fichero. Si el fichero no existe, se crea.

t: tipo texto, si no se especifica "t" ni "b", se asume por defecto que es "t"

b: tipo binario.

10.3. Cerrar un archivo

Función fclose:

Sintaxis:

int fclose(FILE *fichero); Es importante cerrar los ficheros abiertos antes de abandonar la aplicación. Esta función sirve para eso. Cerrar un fichero almacena los datos que aún están en el buffer de memoria, y actualiza algunos datos de la cabecera del fichero que mantiene el sistema operativo. Además permite que otros programas puedan abrir el fichero para su uso. Muy a menudo, los ficheros no pueden ser compartidos por varios programas.

Un valor de retorno cero indica que el fichero ha sido correctamente cerrado, si ha habido algún error, el valor de retorno es la constante EOF. El parámetro es un puntero a la estructura FILE del fichero que queremos cerrar.

10.4. Escribir en un archivo

A continuación se da una serie de instrucciones para la escritura de datos en un archivo.

Función fprintf: Sintaxis:

int fprintf(FILE *fichero, const char *formato, ...);

La función fprintf funciona igual que printf en cuanto a parámetros, pero la salida se dirige a un fichero en lugar de a la pantalla.

10.4.1. Función fputc:

Sintaxis:

int fputc(int caracter, FILE *fichero); Esta función escribe un carácter a un fichero.

El valor de retorno es el carácter escrito, si la operación fue completada con éxito, en caso contrario será EOF. Los parámetros de entrada son el carácter a escribir, convertido a int y un puntero a una estructura FILE del fichero en el que se hará la escritura.

10.4.2. Ejemplo de la función Rosa

Escribir el código para graficar la función polar de la rosa de 6 petalos

```
/*
 * File:   funcion_rosa.c
 * Author: felix
 *
 * Created on 7 de diciembre de 2011, 10:50
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void ejemplo() {
    FILE *fp;
    double x, y, theta, r, a = 2, k = 6, phi = 0;

    fp = fopen("salida.txt","w");

    for(theta =0; theta <= 2*3.1416; theta += 0.01) {
        // Formula Polar para la Rosa
        r = a*cos(k*theta + phi);

        // Conversion a rectangular
```

```
        x = r*cos(theta);
        y = r*sin(theta);

        fprintf(fp, "%f, %f\n", x, y);
    }

    fclose(fp) ;
}

int main(int argc, char** argv) {
    ejemplo();
    return (EXIT_SUCCESS);
}
```

10.4.3. Función fputs:

Sintaxis:

int fputs(const char *cadena, FILE *stream);

La función fputs escribe una cadena en un fichero. No se añade el carácter de retorno de línea ni el carácter nulo final.

El valor de retorno es un número no negativo o EOF en caso de error. Los parámetros de entrada son la cadena a escribir y un puntero a la estructura FILE del fichero donde se realizará la escritura.

10.4.4. Función fwrite:

Sintaxis:

size_t fwrite(void *puntero, size_t tamaño, size_t nregistros, FILE *fichero);

Esta función también esta pensada para trabajar con registros de longitud constante y forma pareja con fread. Es capaz de escribir hacia un fichero uno o varios registros de la misma longitud almacenados a partir de una dirección de memoria determinada.

El valor de retorno es el número de registros escritos, no el número de bytes. Los parámetros son: un puntero a la zona de memoria donde se almacenarán los datos leídos, el tamaño de cada registro, el número de registros a leer y un puntero a la estructura FILE del fichero del que se hará la lectura.

Ejemplo:

```
// copia.c: Copia de ficheros
// Uso: copia <fichero_origen> <fichero_destino>

#include <stdio.h>

int main(int argc, char **argv) {
    FILE *fe, *fs;
    unsigned char buffer[2048]; // Buffer de 2 Kbytes
    int bytesLeidos;

    if(argc != 3) {
       printf("Usar: copia <fichero_origen> <fichero_destino>\n");
       return 1;
    }

    // Abrir el fichero de entrada en lectura y binario
    fe = fopen(argv[1], "rb");
    if(!fe) {
       printf("El fichero %s no existe o no puede ser abierto.\n", argv[1]);
       return 1;
    }
    // Crear o sobreescribir el fichero de salida en binario
    fs = fopen(argv[2], "wb");
    if(!fs) {
       printf("El fichero %s no puede ser creado.\n", argv[2]);
       fclose(fe);
       return 1;
    }
    // Bucle de copia:
    while((bytesLeidos = fread(buffer, 1, 2048, fe)))
       fwrite(buffer, 1, bytesLeidos, fs);
    // Cerrar ficheros:
    fclose(fe);
    fclose(fs);
    return 0;
}
```

10.5. Leer un archivo

A continuación se da una serie de funciones que permiten leer datos de un archivo.

10.5.1. Función fscanf:

Sintaxis:

int fscanf(FILE *fichero, const char *formato, ...); La función fscanf funciona igual que scanf en cuanto a parámetros, pero la entrada se toma de un fichero en lugar del teclado.

10.5.2. Ejemplo

El siguiente es un ejemplo de escritura en un archivo de un valor entero y una cadena de texto. Note que la cadena de texto tiene números.

```
/*
 * File:   lee_archivo.c
 * Author: felix
 *
 * Created on 6 de diciembre de 2011, 13:11
 */

#include <stdio.h>
#include <stdlib.h>

/*
 * Escribe en un archivo y le los datos colocados en el
 */

int main(){

    FILE *fp;

    fp = fopen("test.txt","a");
    fprintf(fp,"%d %s %d\n",1,"1 a 1",2);
    fclose(fp) ;

    fp = fopen("test.txt","r");
```

```
    char texto[10];
    int a, b;
    fscanf(fp,"%d %s %d\n",&a, texto, &b);
    printf(" %d         %s    %d\n",a, texto, b);


    fclose(fp);
    return 0;
}
```

10.5.3. Lectura de una matriz

El siguiente código muestra como leer los datos de una matriz almacenada en un archivo. Suponemos que en el primer renglón esta la información de los renglones y la columnas y después la información de los datos por renglones. Los datos de la matriz son números reales.

```
/*
 * File:   Lee_Matriz_archivo.c
 * Author: felix
 *
 * Created on 7 de noviembre de 2011, 18:55
 */

#include <stdio.h>
#include <stdlib.h>

typedef float TIPO;
void Imprime_Matriz(TIPO **a, int ren, int col);
TIPO **Lee_Matriz(char *archivo, int *ren, int * col);
TIPO ** Matriz(int ren, int col);

void Imprime_Matriz(TIPO **a, int ren, int col){
  int i, j;

  for(i=0; i<ren; i++) {
    for (j = 0; j < col; j++)
      printf("%f ", a[i][j]);
    printf("\n");
  }
}
```

```
TIPO **Lee_Matriz(char *archivo, int *ren, int *col) {
    FILE *fp;
    int i, j, nr, nc;
    TIPO aux;
    TIPO ** A;

    if((fp = fopen(archivo, "r"))==NULL){
        printf("No pude abrir el archivo\n");
        exit(0);
    }

    fscanf(fp, "%d %d", &nr, &nc);
    printf("Matriz de tamanio ren = %d col = %d\n", nr, nc);
    *ren = nr;
    *col = nc;

    A = Matriz(nr, nc);

    for(i=0; i<nr; i++) {
        for(j=0; j<nc; j++) {
            fscanf(fp, "%f", &aux);
            A[i][j] = aux;
        }
    }

    fclose(fp);
    return A;
}

TIPO ** Matriz(int ren, int col){
  TIPO **aux;
  int i, j;

  aux = (TIPO **) calloc(ren, sizeof(TIPO*)); // buscar en stdlib.h

  for(i=0; i<ren; i++)
aux[i] = (TIPO *) calloc(col, sizeof(TIPO));

  return aux;
}
```

```
int main(int argc, char** argv) {
    int ren, col;
    TIPO** A;

    A = Lee_Matriz("prueba.txt", &ren, &col);

    Imprime_Matriz(A, ren, col);

    return (EXIT_SUCCESS);
}
```

10.5.4. Función fgetc:

Sintaxis:

int fgetc(FILE *fichero); Esta función lee un carácter desde un fichero.

El valor de retorno es el carácter leído como un unsigned char convertido a int. Si no hay ningún carácter disponible, el valor de retorno es EOF. El parámetro es un puntero a una estructura FILE del fichero del que se hará la lectura.

10.5.5. Función fgets:

Sintaxis:

char *fgets(char *cadena, int n, FILE *fichero); Esta función está diseñada para leer cadenas de caracteres. Leer· hasta n-1 caracteres o hasta que lea un retorno de línea. En este último caso, el carácter de retorno de línea también es leído.

El parámetro n nos permite limitar la lectura para evitar desbordar el espacio disponible en la cadena.

El valor de retorno es un puntero a la cadena leÌda, si se leyó con éxito, y es NULL si se detecta el final del fichero o si hay un error. Los parámetros son: la cadena a leer, el número de caracteres máximo a leer y un puntero a una estructura FILE del fichero del que se leer·.

10.5.6. Función fread:

Sintaxis:

size_t fread(void *puntero, size_t tamaño, size_t nregistros, FILE *fichero); Esta función está pensada para trabajar con registros de longitud constante. Es capaz de leer desde un fichero uno o varios registros de la misma longitud y a partir de una dirección de memoria determinada. El usuario es responsable de asegurarse de que hay espacio suficiente para contener la información leída.

El valor de retorno es el número de registros leídos, no el número de bytes. Los parámetros son: un puntero a la zona de memoria donde se almacenarán los datos leídos, el tamaño de cada registro, el número de registros a leer y un puntero a la estructura FILE del fichero del que se hará la lectura.

10.5.7. Lectura de un archivo en binario

El siguiente ejemplo muestra el uso de fwrite y fread en un archivo binario. Note que así como se almacena la información en la función Escribe_Archivo() se debe leer en la función Lee_Archivo().

```
/*
 * File:   archivo_binario.c
 * Author: felix
 *
 * Created on 6 de diciembre de 2011, 16:54
 */

#include <stdio.h>
#include <stdlib.h>

/* Our structure */
struct reg
{
    int x,y,z;
};

void Escribe_Archivo();
void Lee_Archivo();

/*
```

```
 * Funcion para leer los datos que se escribieron en un archivo
 * Note que los valores se leen en la misma forma que se escriben
 */

void Lee_Archivo(){
    int i;
    FILE *ptr_archivo;
    struct reg mi_registro;
    float a;
    char c[11];

    ptr_archivo=fopen("prueba.bin","rb");

    if (!ptr_archivo){
        printf("No es posible abrir el archivo!");
        return ;
    }

    for (i=1; i<= 10; i++) {
fread(&mi_registro, sizeof(struct reg), 1, ptr_archivo);

        printf("%d %d %d\n", mi_registro.x, mi_registro.y, mi_registro.z);
    }

    fread(&a, sizeof(float), 1, ptr_archivo);
    fread(&i, sizeof(int), 1, ptr_archivo);
    fread(c, sizeof(char), 11, ptr_archivo);

    printf("%f, %d, %s\n", a, i, c);

    fclose(ptr_archivo);
}

/*
 * Funcion para escribir los datos en el archivo de salida
 * Los datos deben ser leidos en la misma forma que se escriben
 */

void Escribe_Archivo(){
    int i;
    FILE *ptr_archivo;
```

```
    struct reg mi_registro;
    float a = 100.4;
    char c[11]= "Hola Mundo";

    ptr_archivo=fopen("prueba.bin","wb");

    if (!ptr_archivo){
        printf("No es posible abrir el archivo!");
        return ;
    }

    for (i=1; i<= 10; i++){
mi_registro.x= i;
fwrite(&mi_registro, sizeof(struct reg), 1, ptr_archivo);
    }

    fwrite(&a, sizeof(float), 1, ptr_archivo);

    fwrite(&i, sizeof(int), 1, ptr_archivo);
    fwrite(c, sizeof(char), 11, ptr_archivo);

    fclose(ptr_archivo);
}

int main()
{
    Escribe_Archivo();
    Lee_Archivo();
    return 0;
}
```

ver [?]

10.6. Otras funciones para manejo de archivos

10.6.1. Función feof:

Sintaxis:

int feof(FILE *fichero);

Esta función sirve para comprobar si se ha alcanzado el final del fichero. Muy frecuentemente deberemos trabajar con todos los valores almacenados en un archivo de forma secuencial, la forma que suelen tener los bucles para leer todos los datos de un archivo es permanecer leyendo mientras no se detecte el fin de fichero. Esta función suele usarse como prueba para verificar si se ha alcanzado o no ese punto.

El valor de retorno es distinto de cero solo si no se ha alcanzado el fin de fichero. El parámetro es un puntero a la estructura FILE del fichero que queremos verificar.

10.6.2. Función rewind:

Sintaxis:

void rewind(FILE *fichero);

Es una función heredada de los tiempos de las cintas magnéticas. Literalmente significa "rebobinar", y hace referencia a que para volver al principio de un archivo almacenado en cinta, había que rebobinarla. Eso es lo que hace ésta función, sitúa el cursor de lectura/escritura al principio del archivo.

El parámetro es un puntero a la estructura FILE del fichero que queremos rebobinar.

Ejemplos:

```
// ejemplo1.c: Muestra un fichero dos veces.
#include <stdio.h>

int main() {
   FILE *fichero;

   fichero = fopen("ejemplo1.c", "r");
   while(!feof(fichero)) fputc(fgetc(fichero), stdout);
   rewind(fichero);
   while(!feof(fichero)) fputc(fgetc(fichero), stdout);
   fclose(fichero);
   getchar();
   return 0;
}
```

10.6.3. Función fflush:

Sintaxis:

int fflush(FILE *fichero);

Esta función fuerza la salida de los datos acumulados en el buffer de salida del fichero. Para mejorar las prestaciones del manejo de ficheros se utilizan buffers, almacenes temporales de datos en memoria, las operaciones de salida se hacen a travéss del buffer, y solo cuando el buffer se llena se realiza la escritura en el disco y se vacía el buffer. En ocasiones nos hace falta vaciar ese buffer de un modo manual, para eso sirve ésta función.

El valor de retorno es cero si la función se ejecutó con éxito, y EOF si hubo algún error. El parámetro de entrada es un puntero a la estructura FILE del fichero del que se quiere vaciar el buffer. Si es NULL se hará el vaciado de todos los ficheros abiertos.

10.6.4. Función fseek:

Sintaxis:

int fseek(FILE *fichero, long int desplazamiento, int origen);

Esta función sirve para situar el cursor del fichero para leer o escribir en el lugar deseado.

El valor de retorno es cero si la función tuvo éxito, y un valor distinto de cero si hubo algún error.

Los parámetros de entrada son: un puntero a una estructura FILE del fichero en el que queremos cambiar el cursor de lectura/escritura, el valor del desplazamiento y el punto de origen desde el que se calculará el desplazamiento.

El parámetro origen puede tener tres posibles valores:

SEEK_SET el desplazamiento se cuenta desde el principio del fichero. El primer byte del fichero tiene un desplazamiento cero. SEEK_CUR el desplazamiento se cuenta desde la posición actual del cursor. SEEK_END el desplazamiento se cuenta desde el final del fichero.

10.6.5. Función ftell:

Sintaxis:

long int ftell(FILE *fichero);

La función ftell sirve para averiguar la posición actual del cursor de lecturaescritura de un fichero.

El valor de retorno será esa posición, o -1 si hay algún error.

El parámetro de entrada es un puntero a una estructura FILE del fichero del que queremos leer la posición del cursor de lecturaescritura.

10.7. Aplicaciones

10.7.1. Lectura de algunos tipos de datos

Este ejemplo muestra como leer desde un archivo, utilizando como terminado la función feof(), caracteres, números flotantes y palabras.

```
/*
 * File:   lee_datos.c
 * Author: felix
 *
 * Created on 14 de diciembre de 2011, 18:08
 */

#include <stdio.h>
#include <stdlib.h>

/*
 * Estas funciones permiten leer datos de archivos en diferentes formatos
 */

void Lee_flotantes(char *nombre) {
    FILE *fp;
    char c[100];
    float f;

    printf("Nombre del archivo %s\n", nombre);

    fp = fopen(nombre, "r");

    if(!fp) {
        printf("No pude abrir el archivo\n");
        return;
    }

    while(!feof(fp)) {
        fscanf(fp, "%f,", &f);
```

```
        if(feof(fp)) break;
        printf("%f\n", f);
    }

    fclose(fp);

    return;
}

int Lee_Palabras(char *nombre) {
    FILE *fp;
    char c[100];

    printf("Nombre del archivo %s\n", nombre);

    fp = fopen(nombre, "rb");

    if(!fp) {
        printf("No pude abrir el archivo\n");
        return 0;
    }

    while(!feof(fp)) {
        fscanf(fp, "%s,", c);
        if(feof(fp)) break;
        printf("%s\n", c);
    }

    fclose(fp);

    return 0;
}

int Lee_Caracteres(char *nombre) {
    FILE *fp;
    char c;

    printf("Nombre del archivo %s\n", nombre);

    fp = fopen(nombre, "rb");
```

```
    if(!fp) {
        printf("No pude abrir el archivo\n");
        return 0;
    }

    printf("Comienzo a leer \n");

    while(!feof(fp)) {
        c = fgetc(fp);
        if(feof(fp)) break;
        printf("%c ", c);
    }

    fclose(fp);

    return 0;
}

int main(int argc, char** argv) {
    Lee_Caracteres("entrada.txt");
    //Lee_Palabras("entrada.txt");
    //Lee_flotantes("salida.txt");
    return (EXIT_SUCCESS);
}
```

10.7.2. Código

El siguiente es un ejemplo de la codificación de un archivo de texto. Las letras son cambiadas de acuerdo con lo siguiente:

Si el carácter leído es a pone z, si es b pone x, y así sucesivamente hasta z pone a. Para los números lo hace de manera muy similar 0-9, 1-8, 2-7, etc.

```
/*
 * File:   codigo.c
 * Author: felix
 *
 * Created on 14 de diciembre de 2011, 18:08
 */

#include <stdio.h>
```

```
#include <stdlib.h>

/*
 * Estas funciones permiten leer datos de archivos en diferentes formatos
 */

char codigo(char a) ;
int Lee_Caracteres(char *entrada, char *salida);

int Lee_Caracteres(char *entrada, char *salida) {
    FILE *fp1, *fp2;
    char c;

    fp1 = fopen(entrada, "r");
    fp2 = fopen(salida, "w");

    if(!fp1) {
        printf("No pude abrir el archivo %s\n", entrada);
        return 0;
    }

    while(!feof(fp1)) {
        c = fgetc(fp1);
        if(feof(fp1)) break;
        fprintf(fp2, "%c", codigo(c));
    }

    fclose(fp1);
    fclose(fp2);

    return 0;
}

char codigo(char a) {
   if(a >= 'a' && a <= 'z') return ('a'+'z' -a);
   if(a >= 'A' && a <= 'Z') return ('A' + 'Z' -a);
   if(a >= '0' && a <= '9') return ('0'+'9' -a);
   return a;
}

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

```
    Lee_Caracteres("entrada.txt", "salida.txt");

    return (EXIT_SUCCESS);
}
```

10.7.3. Cambia orden de la palabras

El siguiente es un ejemplo de lectura de un archivo. Las palabras leídas se cambia el orden de las letras si cambiar la primer letra y la última y se imprime. Lo sorprendente es que el resultado es un texto que se puede leer.

```
/*
 * File:   palabras.c
 * Author: felix
 *
 * Created on 21 de junio de 2012, 20:38
 */

#include <stdio.h>
#include <stdlib.h>
#include <strings.h>

void cambia_orden(char *a) {
    int i, k, l, N;
    int aux;

    N = strlen(a);

    if(N==2) return;

    for(i=0; i<2*N; i++) {
        k = numero_aleatorio(N);
        l = numero_aleatorio(N);

        aux = a[k];
        a[k] = a[l];
        a[l] = aux;
    }
}

int numero_aleatorio(int N)
```

```
{
   int n;
   n = 1 + rand() % (N-2);
   return(n);
}

int Lee_Palabras(char *nombre) {
    FILE *fp;
    char c[100];

    printf("Nombre del archivo %s\n", nombre);

    fp = fopen(nombre, "rb");

    if(!fp) {
        printf("No pude abrir el archivo\n");
        return 0;
    }

    while(1) {
        fscanf(fp, "%s,", c);
        if(feof(fp)) break;
        cambia_orden(c);
        printf("%s ", c);
    }

    fclose(fp);

    return 0;
}

int main(int argc, char** argv) {

    Lee_Palabras("entrada.txt");

    return (EXIT_SUCCESS);
}
```

10.7.4. Lectura de Imágenes en formato PPM

Una imagen en formato consiste de lo siguiente:

1. Un numero mágico que identifica el tipo de archivo. Un número mágico son los caracteres P6.
2. El ancho y alto de la imagen en formato ASCII separados por un espacio
3. El color máximo de la imagen, normalmente 255
4. Un vector con los colores correspondientes a cada punto de la imagen comenzando en la esquina superior izquierda y terminando en la esquina inferior derecha

```
/*
 * File:   imagen_ppm.c
 * Author: felix
 *
 * Created on 6 de diciembre de 2011, 17:44
 */

#include <stdio.h>
#include <stdlib.h>

void Esfera_Imagen();
void Guarda_Imagen(double ***I, int dimx, int dimy);
double ***Imagen(int N, int M);
double ***Imagen(int N, int M);
double *** Lee_Imagen(char *nombre, int *nr, int *nc);
void Lee_linea(FILE *fp);

/*
 * Crea una imagen de una esfera
 */

void Esfera_Imagen() {
  const int dimx = 800, dimy = 800;
  int i, j;
  FILE *fp = fopen("first.ppm", "wb"); /* b - binary mode */
  static unsigned char color[3];
  int x, y;

  fprintf(fp, "P6\n");
```

```
  fprintf(fp, "# Imagen creada por mi\n");
  fprintf(fp, "%d %d\n", dimx, dimy);
  fprintf(fp, "255\n");

  for (j = 0; j < dimy; ++j)
  {
    for (i = 0; i < dimx; ++i) {
      x = i - dimx/2;
      y = j - dimy/2;

      if(x*x + y*y < 100000) {
          color[0] = 255 - (x*x + y*y) /500;  /* red */
          color[1] = 255 - (x*x + y*y) /500;  /* green */
          color[2] = 255 - (x*x + y*y) /500;  /* blue */
      }
      else {
          color[0] = 0;
          color[1] = 0;
          color[2] = 0;
      }
      (void) fwrite(color, 1, 3, fp);
      }
  }
  fclose(fp);
}
/*
 * Guarda una Imagen
 */

void Guarda_Imagen(double ***I, int dimx, int dimy) {
  int i, j, k;
  FILE *fp = fopen("salida.ppm", "wb"); /* b - binary mode */
  static unsigned char color[3];
  static unsigned char aux;

  fprintf(fp, "P6\n");
  fprintf(fp, "# Imagen creada por mi\n");
  fprintf(fp, "%d %d\n", dimx, dimy);
  fprintf(fp, "255\n");

  for (j = 0; j < dimy; j++){
```

```
    for (i = 0; i < dimx; i++) {
        for(k=0; k<3; k++) {
            aux = (unsigned char) I[k][i][j] %255;
            color[k] = aux;
        }
      fwrite(color, 1, 3, fp);
    }
  }
  fclose(fp);
}

/*
 * Reserva la memoria necesaria para almacenar una imagen en Color
 */

double ***Imagen(int N, int M)
{
    double ***a;
    int i, k;

    a = (double ***) calloc(3, sizeof(double**));

    for(k=0; k<3; k++)
        a[k] = (double **) calloc (N, sizeof(double*));

    for(k=0; k<3; k++)
        for(i=0; i<N; i++)
            a[k][i] = (double *) calloc(M, sizeof (double));
    return a;
}

/*
 * Lee una linea del archivo indicado por el apuntador fp
 */

void Lee_linea(FILE *fp) {
    char c;

    do{
        fscanf(fp,"%c", &c);
        printf("%c", c);
```

```
    } while(c != '\n');
}

/*
 *  Lee una imagen desde archivo
 */

double *** Lee_Imagen(char *nombre, int *nr, int *nc) {
    FILE *fp;
    int dimx, dimy;
    int i, j, k;
    static unsigned char color[3];
    double ***I;

    fp = fopen(nombre, "rb"); /* b - binary mode */
    if(!fp) {
        printf("Error al abrir el archivo\n");
        return;
    }

    printf("Formato del archivo PPM: ");
    Lee_linea(fp);
    printf("Creado por: ");
    Lee_linea(fp);

    fscanf(fp, "%d %d", &dimx, &dimy);
    printf("Renglones %d Columnas %d\n", dimy, dimx);
    *nr = dimx;
    *nc = dimy;

    I = Imagen(dimx, dimy);

    for(j=0; j<dimy; j++)
        for(i=0; i<dimx; i++){
            fread(color, 1, 3, fp);
            for(k=0; k<3; k++)
                I[k][i][j] = color[k];
        }
    fclose(fp);

    return I;
```

```
}

void Ejemplo01() {
    double ***I;
    int nr, nc;
    int i, j, k;
    I = Lee_Imagen("lena2.ppm", &nr, &nc);

    for(k=0; k<3; k++)
        for(i=0; i<nr; i++)
            for(j=0; j<nc; j++)
                I[k][i][j] = 255 - I[k][i][j];

    Guarda_Imagen(I, nr, nc);
}

void Ejemplo02() {
    double ***I;
    int nr, nc;
    int i, j, k;
    double suma;

    I = Lee_Imagen("lena2.ppm", &nr, &nc);

    for(i=0; i<nr; i++)
        for(j=0; j<nc; j++) {
            suma = 0;
            for(k=0; k<3; k++)
                suma += I[k][i][j];
            for(k=0; k<3; k++)
                I[k][i][j] = suma/3;
        }

    Guarda_Imagen(I, nr, nc);
}

int main(int argc, char** argv) {
    Ejemplo02();

    return (EXIT_SUCCESS);
}
```

Directivas del precompilador

El preprocesador tiene más o menos su propio lenguaje el cual puede ser una herrramienta muy poderosa para el programador. Todas las directivas de preprocesador o comandos inician con un #.

Las ventajas que tiene usar el preprocesador son:

- los programas son más fáciles de desarrollar, - son más fáciles de leer, - son más fáciles de modificar - y el código de C es más transportable entre diferentes arquitecturas de máquinas.

11.1. Directiva #define

El preprocesador también permite configurar el lenguaje. Por ejemplo, para cambiar a las sentencias de bloque de código ... delimitadores que haya inventado el programador como inicio ... fin se puede hacer:

```
#define inicio {
#define fin }
```

Durante la compilación todas las ocurrencias de inicio y fin serán reemplazadas por su correspondiente o delimitador y las siguientes étapas de compilación de C no encontrarán ninguna diferencia.

La directiva #define se usa para definir constantes o cualquier sustitución de macro. Su formato es el siguiente:

```
#define <nombre de macro> <nombre de reemplazo>
```

Por ejemplo:

```
#define FALSO 0
#define VERDADERO !FALSO
```

La directiva #define tiene otra poderosa característica: el nombre de macro puede tener argumentos. Cada vez que el compilador encuentra el nombre de macro, los argumentos reales encontrados en el programa reemplazan los argumentos asociados con el nombre de la macro. Por ejemplo:

```
#define MIN(a,b) (a < b) ? a : b

main()
{
int x=10, y=20;

printf("EL minimo es %d\n", MIN(x,y) );
}
```

Cuando se compila este programa, el compilador sustiuirá la expresión definida por MIN(x,y), excepto que x e y serán usados como los operandos. Así después de que el compilador hace la sustitución, la sentencia printf será ésta:

```
printf("El minimo es %d\n", (x < y) ? x : y);
```

Como se puede observar donde se coloque MIN, el texto será reemplazado por la definición apropiada. Por lo tanto, si en el código se hubiera puesto algo como:

```
x = MIN(q+r,s+t);
```

después del preprocesamiento, el código podría verse de la siguiente forma:

```
x = ( q+r < s+t ) ? q+r : s+t;
```

Otros ejemplos usando #define pueden ser:

```
#define Deg_a_Rad(X) (X*M_PI/180.0)
/* Convierte grados sexagesimales a radianes, M_PI es el valor de pi */
/*    y esta definida en la biblioteca math.h */

#define IZQ_DESP_8 <<8
```

La última macro IZQ_DESP_8 es solamente válida en tanto el reemplazo del contexto es válido, por ejemplo: x = y IZQ_DESP_8.

El uso de la sustitución de macros en el lugar de las funciones reales tiene un beneficio importante: incrementa la velocidad del código porque no se penaliza con una llamada de función. Sin embargo, se paga este incremento de velocidad con un incremento en el tamaño del programa porque se duplica el código.

11.2. Directiva #undef

Se usa #undef para quitar una definición de nombre de macro que se haya definido previamente. El formato general es:

```
#undef <nombre de macro>
```

El uso principal de #undef es permitir localizar los nombres de macros sólo en las secciones de código que los necesiten.

11.3. Directiva #include

La directiva del preprocesador #include instruye al compilador para incluir otro archivo fuente que esta dado con esta directiva y de esta forma compilar otro archivo fuente. El archivo fuente que se leerá se debe encerrar entre comillas dobles o paréntesis de ángulo. Por ejemplo:

```
#include <archivo>
```

```
#include "archivo"
```

Cuando se indica ¡archivo¿se le dice al compilador que busque donde estan los archivos incluidos o "include" del sistema. Usualmente los sistemas con UNIX guardan los archivos en el directorio /usr/include.

Si se usa la forma .ªrchivo.es buscado en el directorio actual, es decir, donde el programa esta siendo ejecutado.

Los archivos incluidos usualmente contienen los prototipos de las funciones y las declaraciones de los archivos cabecera (header files) y no tienen código de C (algoritmos).

11.4. Directiva #if Inclusión condicional

La directiva #if evalua una expresión constante entera. Siempre se debe terminar con #endif para delimitir el fin de esta sentencia.

Se pueden así mismo evaluar otro código en caso se cumpla otra condición, o bien, cuando no se cumple ninguna usando #elif o #else respectivamente.

Por ejemplo,

```
#define MEX 0
#define EUA 1
#define FRAN 2

#define PAIS_ACTIVO MEX

#if PAIS_ACTIVO == MEX
char moneda[]="pesos";
#elif PAIS_ACTIVO == EUA
char moneda[]="dolar";
#else
char moneda[]="franco";
#endif
```

Otro método de compilación condicional usa las directivas #ifdef (si definido) y #ifndef (si no definido).

El formato general de #ifdef es:

```
#ifdef <nombre de macro>
<secuencia de sentecias>
#endif
```

Si el nombre de macro ha sido definido en una sentencia #define, se compilará la secuencia de sentecias entre el #ifdef y #endif.

El formato general de #ifdef es:

```
#ifndef <nombre de macro>
<secuencia de sentecias>
#endif
```

Las directivas anteriores son útiles para revisar si las macros están definidas – tal vez por módulos diferentes o archivos de cabecera.

Por ejemplo, para poner el tamaño de un entero para un programa portable entre TurboC de DOS y un sistema operativo con UNIX, sabiendo que TurboC usa enteros de 16 bits y UNIX enteros de 32 bits, entonces si se quiere compilar para TurboC se puede definir una macro TURBOC, la cual será usada de la siguiente forma:

```
#ifdef TURBOC
#define INT_SIZE 16
#else
#define INT_SIZE 32
#endif
```

11.5. Control del preprocesador del compilador

Se puede usar el compilador gcc para controlar los valores dados o definidos en la línea de comandos. Esto permite alguna flexibilidad para configurar valores ademas de tener algunas otras funciones útiles. Para lo anterior, se usa la opción -Dmacro[=defn], por ejemplo:

gcc -DLONGLINEA=80 prog.c -o prog

que hubiese tenido el mismo resultado que

#define LONGLINEA 80

en caso de que hubiera dentro del programa (prog.c) algún #define o #undef pasaría por encima de la entrada de la línea de comandos, y el compilador podría mandar un "warning" sobre esta situación.

También se puede poner un símbolo sin valor, por ejemplo:

gcc -DDEBUG prog.c -o prog

En donde el valor que se toma es de 1 para esta macro.

Las aplicaciones pueden ser diversas, como por ejemplo cuando se quiere como una bandera para depuración se puede hacer algo como lo siguiente:

```
#ifdef DEBUG
printf("Depurando: Versión del programa 1.0\n");
#else
printf("Version del programa 1.0 (Estable)\n");
#endif
```

Como los comandos del preprocesador pueden estar en cualquier parte de un programa, se pueden filtrar variables para mostrar su valor, cuando se esta depurando, ver el siguiente ejemplo:

```
x = y * 3;

#ifdef DEBUG
    printf("Depurando: variables x e y iguales a \n",x,y);
#endif
```

La opción -E hace que la compilación se detenga después de la étapa de preprocesamiento, por lo anterior no se esta propiamente compilando el programa. La salida del preprocesamiento es enviada a la entrada estándar. GCC ignora los archivos de entrada que no requieran preprocesamiento.

11.6. Otras directivas del preprocesador

La directiva #error forza al compilador a parar la compilación cuando la encuentra. Se usa principalmente para depuración. Por ejemplo:

```
#ifdef OS_MSDOS
    #include <msdos.h>
#elifdef OS_UNIX
    #include "default.h"
#else
    #error Sistema Operativo incorrecto
#endif
```

La directiva #line número "cadena" informa al preprocesador cual es el número siguiente de la línea de entrada. Cadena es opcional y nombra la siguiente línea de entrada. Esto es usado frecuentemente cuando son traducidos otros lenguajes a C. Por ejemplo, los mensajes de error producidos por el compilador de C pueden referenciar el nombre del archivo y la línea de la fuente original en vez de los archivos intermedios de C.

Por ejemplo, lo siguiente especifica que el contador de línea empezará con 100.

```
#line 100 "test.c"    /* inicializa el contador de linea y nombre de archivo */
main()                /* linea 100 */
{
    printf("%d\n",__LINE__); /* macro predefinida, linea 102 */
    printf("%s\n",__FILE__); /* macro predefinida para el nombre */
}
```

Tareas

12.1. Calculo de Porcentaje

Al realizar la compra de un artículo, me pidieron como anticipo $135.00 equivalente al 25 % de adelanto. Sin embargo olvide el precio final y no se cuanto debo pagar para recoger el articulo. Escribir un programa en C que permita calcular el precio total del articulo y la cantidad total que adeudo.

12.2. Conversión de Tipos

Hacer una función que permita hacer el redondeo de un número flotante a dos cifras significativas.

12.3. Expresiones Lógicas

Hacer un programa que verifique si en el intervalo $[-2, 3]$ se cumple que la desigualdad $(x + 2) * (x - 3) > 0$.

12.4. Entrada Salida

Hacer un programa, que reciba tres números de punto flotante y calcule su promedio

12.5. Problema de Física

Un bloque de masa $m_1 = 3{,}0$ slugs descansa en un plano liso que està inclinado un ángulo de 30 grados respecto a la horizontal y está unido, por medio de una cuerda que para por una pequeña polea sin fricción, con un segundo bloque de masa $m_2 = 2$ slugs suspendido verticalmente tal como se muestra en la figura 12.4. (a) ¿Cual es la aceleración de cada cuerpo? b) ¿Cuál es la tensión de la cuerda?. c) Repita el problema con diferentes valores de inclinación del plano

Escriba el código correspondiente en C para solucionar este problema

Respuesta a.- 3.2 pies/seg2 y b.- 58 lb

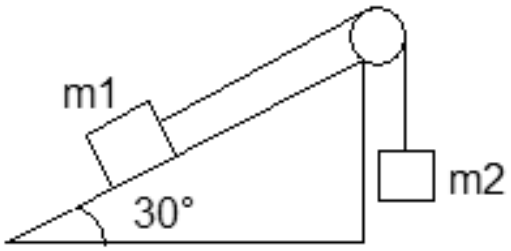


Figura 12.4: Ejemplo de plano inclinado.

12.6. Condicionales

Plante un problema y escriba un programa que solucione un problema utilizando condicionales

12.7. Dias del año transcurridos

Dada una fecha valida de un año, calcular cuantos días del año han transcurrido.

12.8. Ciclos

Escriba un programa que imprima las siguientes secuencias de números, utilizando en los tres casos ciclos for, while y do-while

- 10, 20, 30, 40 ...
- -7, -3, 1, 5, ...
- 25, 16, 9, 4, 1, 0, 1, 4, 16, 25

12.9. Escritura de funciones

1. Escribir una función que permita sumar dos horas validas en formato valido y el resultado sea una hora en formato valido 2.- Escribir una función para calcular los días transcurridos entre dos fechas validas

12.10. Función Coseno

Modificar los códigos del ejemplo anterior para implementar la función coseno

12.11. Calculadora recursiva

Hacer un conjunto de funciones que realice las cuatro operaciones básicas de suma, resta, multiplicación y división utilizando recursividad. Estas funciones deber·n estar definidas en un archivo diferente a donde se localiza la función main con el objeto de hacer programación modular.

12.12. Tablas de multiplicar

Escribir una función que almacene en un arreglo bidimensional las tablas de multiplicar. La función recibirá el número y regresara el arreglo lleno.

12.13. Matrices elevadas a una potencia entera

Hacer una función que reciba una matriz y regrese otra matriz con la información de la matriz dada elevada a un entero dado

12.14. Árbol de Navidad

Hacer un programa que permita formar un pino de navidad, utilizando las letras de la palabra feliz Navidad. Utilice ciclos y las operaciones dadas para manejo de cadenas.

12.15. Estructuras

Crea una estructura de datos que maneje la información de un directorio telefónico. El programa debe tener la posibilidad de buscar el número telefónico por el nombre de una persona.