Principios de Programación

Introducción a la Programación

1. Conceptos de Computación

1.1. Qué es una computadora

Una computadora es un dispositivo capaz de ejecutar cálculos y tomar decisiones lógicas a velocidades millones y miles de millones de veces más rápido de lo que pueden hacerlo los seres humanos.

Las computadoras procesan datos bajo el control de un conjunto de instrucciones que se conocen como *programas de computación*. Estos programas guían a la computadora a través de conjuntos ordenados de acciones especificados por personas a las que se conoce como *programadores*.

Los varios dispositivos (teclado, pantalla, discos, memoria, unidades procesadoras) se conocen como *hardware*. Los programas de computación se conocen como el *software*.

1.2. Lenguajes de programación

Los programadores escriben instrucciones en diferentes lenguajes de programación, algunos comprensibles en forma directa por la computadora y otros que requieren pasos intermedios de traducción.

Existen hoy día cientos de lenguajes de computadora. Estos pueden ser categorizados en tres tipos generales:

1. Lenguajes de máquina
2. Lenguajes ensambladores
3. Lenguajes de alto nivel

Cualquier computadora solo puede entender directamente su propio *lenguaje de máquina*. Este está relacionado intimamente con el diseño del hardware de la computadora. Estos lenguajes son dependientes de la máquina. En general consisten de cadenas de números que instruyen a las computadoras para que ejecuten sus operaciones más elementales una a la vez.

ejemplo:

1300042774
1400593419
1200274027

Conforme las computadoras se hicieron más populares se hizo aparente que la programación en lenguaje de máquina era lenta y tediosa para los programadores. Se empezaron entonces a utilizar abreviaturas similares al ingles para representar las operaciones elementales de la computadora. Estas abreviaturas formaron la base de los *lenguajes ensambladores*. Se desarrollaron *programas de traducción* para convertir los programas del lenguaje ensamblador a lenguaje de máquina.

ejemplo:

```
LOAD BASE
ADD VALOR
STORE RESULTADO
```

La utilización de las computadoras aumentó con rapidez con la llegada de los lenguajes ensambladores pero aún con estos se necesitaban muchas instrucciones para llevar a cabo inclusive tareas sencillas. Para acelerar el proceso de programación se desarrollaron *lenguajes de alto nivel* en los cuales se pueden escribir sentencias simples que llevan a cabo tareas sustanciales.

2. Los fundamentos del entorno de C

Todos los sistemas C consisten en general de tres partes: el entorno, el lenguaje y la biblioteca estándar.

Veamos el entorno típico de desarrollo de C:

1. Fase 1 : Editor. El programa es creado en el editor y almacenado en disco.
2. Fase 2 : Preprocesador. El programa preprocesador procesa el código. Se ejecuta antes de la traducción. Se encarga de la inclusión de otros archivos en el archivo a compilar y en el reemplazo de símbolos especiales con texto de programa. Es invocado por el compilador antes de la traducción del programa a lenguaje de máquina.
3. Fase 3 : Compilador. El compilador crea el código objeto y lo almacena en disco.
4. Fase 4 : Enlazador. El enlazador vincula el código objeto con las bibliotecas, crea el archivo a.out y lo almacena en disco.
5. Fase 5 : Cargador. El cargador coloca el programa en memoria. Antes de que un programa pueda ser ejecutado debe ser colocado en memoria.
6. Fase 6 : CPU. La CPU toma cada una de las instrucciones y las ejecuta almacenando posiblemente nuevos valores de datos conforme se ejecuta el programa.

Veremos el lenguaje a lo largo del curso.

Veamos la biblioteca estandar.

Los programas C consisten de módulos que se denominan funciones. En general, además de las funciones que uno programa se utilizan funciones ya existentes que se encuentran en la biblioteca estandar de C.

Utilizar funciones ya existentes se conoce como reutilización de software.

En general utilizaremos:

- Funciones de la biblioteca estandar C
- Funciones que crea uno mismo
- Funciones creadas por otros programadores

3. Algoritmos

Antes de escribir un programa para resolver un problema es esencial tener comprensión completa del mismo y un método planeado de forma cuidadosa para su resolución.

La solución a cualquier problema de cómputo involucra la ejecución de una serie de acciones en un orden especifico. Un *algoritmo* es un procedimiento que resuelve un problema en términos de:

1. las acciones a ejecutarse
2. el orden en el cual estas deben ejecutarse

Veamos un ejemplo, el algoritmo de "levantarse"que debe seguir un ejecutivo para salir de la cama y llegar al trabajo:

Salir de la cama
Quitarse los pijamas
Darse una ducha
Vestirse
Desayunar
Utilizar el vehiculo para llegar al trabajo

Si cambiaramos el orden y colocaramos Vestirse y Darse una ducha intercambiados el ejecutivo llegaría al trabajo mojado.

4. Pseudocódigo

El pseudocódigo es un lenguaje artificial e informal que auxilia a los programadores a desarrollar los algoritmos. Este es similar al lenguaje natural, es amigable aunque no se trate de un lenguaje verdadero de programación de computadoras.

Es un lenguaje intermedio entre el lenguaje natural y los lenguajes de programación.

Los programas en pseudocódigo no son ejecutables, se utilizan porque ayudan al programador a pensar un programa antes de intentar escribirlo en un lenguaje de programación como C.

Un programa preparado cuidadosamente en pseudocódigo puede ser convertido con facilidad en el correspondiente programa en C. Esto se lleva a cabo en muchos casos reemplazando enunciados en pseudocódigo por sus equivalentes en C.

A continuación veremos las ïnstrucciones"que utilizaremos para escribir pseudocódigo:

- Definir constante con valor x
- Definir variable con valor x
- Realizar operaciones aritméticas (escribimos la operación)
- Leer datos
- Imprimir datos
- Si se cumple condicion entonces realizo sentencia
- Si se cumple condicion entonces realizo sentencia1 si no realizo sentencia2
- Mientras se cumpla condicion ejecutar sentencia

5. Diagramas de flujo

Por lo general en un programa los enunciados son ejecutados uno despues de otro, en el orden en que aparecen escritos. Esto se conoce como *ejecución secuencial*.

Varios enunciados de C, que pronto analizaremos le permiten al programador especificar que el enunciado siguiente a ejecutar pueda ser otro diferente del que sigue a continuación.

Un diagrama de flujo es una representación gráfica de un algoritmo. Los diagramas de flujo se trazan utilizando símbolos de uso especial como ser rectángulos y rombos. Estos símbolos están conectados entre si por flechas conocidas como *líneas de flujo*.

Al igual que el pseudocódigo, los diagramas de flujo son útiles para el desarrollo y la representación de algoritmos.

El diagrama de flujo correspondiente a una acción como ser un cálculo o una operación de entrada/salida es el rectángulo.

El diagrama correspondiente a una secuencia es una secuencia de rectangulos. Por ejemplo, el diagrama correspondiente a:

sumar grado a total
sumar 1 a contador

es el de la figura 1.

Cuando dibujamos una porción de un algoritmo colocamos pequeños círculos conocidos como símbolos de conexión.

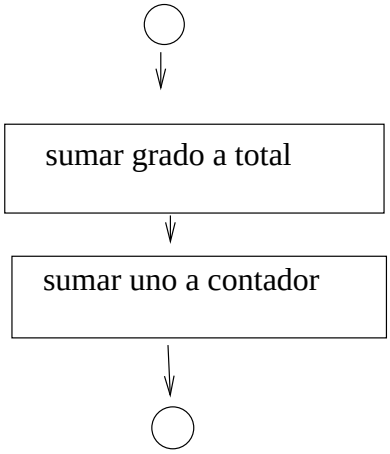

Figura 1: Diagrama de flujo de secuencia

Quiza el simbolo de diagrama de flujo más importante es el rombo, también conocido como *símbolo de decisión*, que indica donde se debe tomar una decisión.

Supongamos tenemos el siguiente pseudocódigo:

Si se cumple grado es mayor o igual a 60 entonces
Imprimir “Aprobado”

el diagrama correspondiente es el de la figura 2.

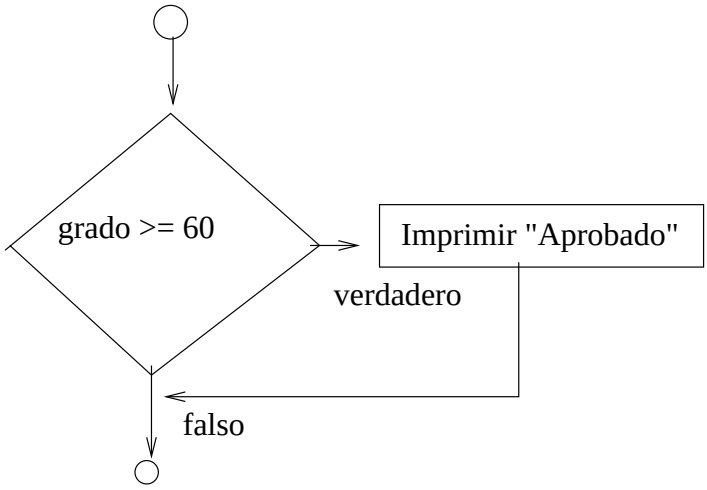

Figura 2: Diagrama de flujo de selección

Consideremos ahora el Si-si no. Este permite especificar que se ejecuten acciones distintas cuando la condición sea verdadera y cuando sea falsa. Dado el siguiente pseudocódigo,

Si se cumple grado es mayor o igual a 60 entonces
   Imprimir “Aprobado”
si no
   Imprimir “Aplazado”

el diagrama correspondiente es el de la figura 3.

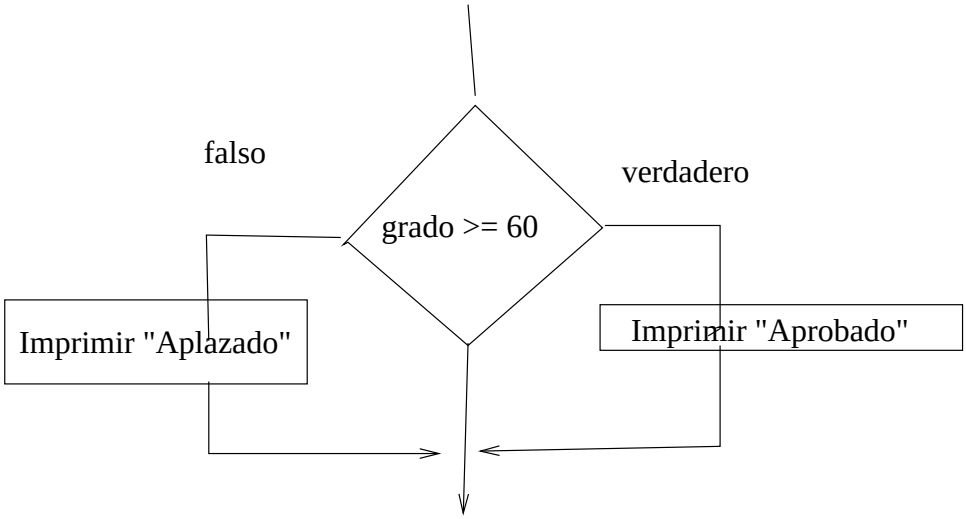

Figura 3: Diagrama de flujo de doble selección

Consideremos ahora el ”Mientras”. Supongamos queremos calcular la primera potencia de 2 superior a 1000. El pseudocódigo es:

Definir producto=2.
Mientras producto <= 1000
        multiplicar producto por 2.
Imprimir producto

el diagrama correspondiente es el de la figura 4.

6. Casos de Estudio (pseudocódigo)

6.1. Repetición controlada por contador

Consideremos el siguiente enunciado:

Una clase de diez alumnos hizo un examen. Las calificaciones (enteros en el rango de 0 a 100) correspondientes a este examen están a su disposición. Determine el promedio de la clase en este examen.

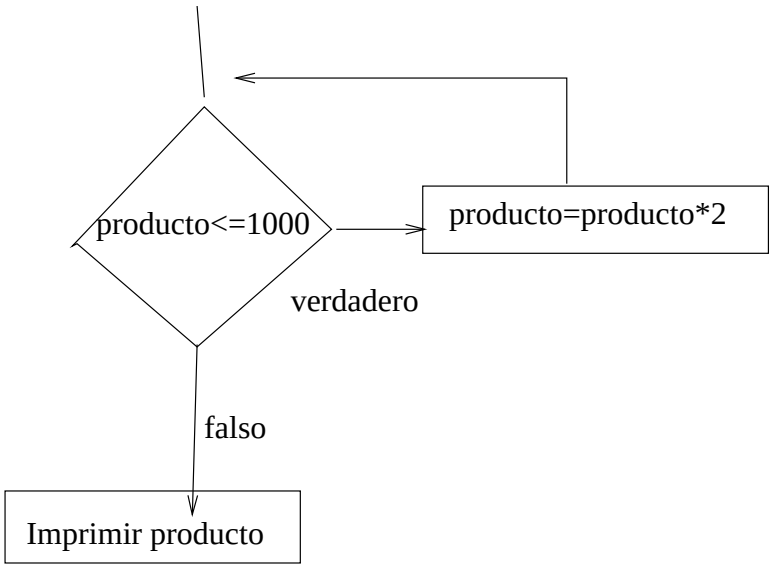

Figura 4: Diagrama de flujo de mientras

El promedio de la clase es igual a la suma de las calificaciones dividida por el número de alumnos. Para resolver el problema debemos leer las calificaciones, calcular el promedio e imprimir el resultado.

El contador que utilizaremos es la cantidad de alumnos, mientras no pasemos de los 10 alumnos leeremos las calificaciones.

Utilizaremos además una variable total en la que sumaremos las calificaciones. Esta variable debe inicializarse en 0.

El pseudocódigo es el siguiente:

```
Inicializar total en 0
Inicializar nro_alumno a 1
Mientras nro_alumno menor o igual a 10
        Leer siguiente calificacion
        Sumar calificacion a total
        Sumar 1 a nro_alumno
Calcular promedio = total /10
Imprimir promedio
```

6.2. Repetición controlada por centinela

El problema es igual al anterior salvo que en vez de considerar 10 estudiantes consideramos una cantidad arbitraria.

Ingresaremos datos y cuando hayamos terminado con todos los datos ingresaremos -1.

El pseudocódigo es el siguiente:

```
Inicializar total en 0
Inicializar cant_alumnos a 0
```

Leer centinela
Mientras centinela distinto de -1
Leer siguiente calificacion
Sumar calificacion a total
Sumar 1 a cant_alumnos
Leer centinela
Si cant_alumnos es distinto de 0 entonces
calcular promedio = total /cant_alumnos
Imprimir promedio
si no Imprimir "no se ingresaron calificaciones"

6.3. Estructuras de control anidadas

Consideremos el siguiente problema:

Una universidad ofrece un curso que prepara alumnos para un examen. La universidad desea saber que tan bien salieron sus alumnos en el examen. Se ingresará un 1 si el alumno pasó el examen y un 2 si lo reprobó. Se quiere saber total de aprobados y total de reprobados en un total de 10 alumnos.

El pseudocódigo es el siguiente:

Inicializar total en 0
Inicializar aprobados en 0
Inicializar reprobados en 0
Inicializar cant_alumnos a 1
Mientras cant_alumnos menor o igual a 10
Leer siguiente calificacion
Si calificacion = 1 sumar 1 a aprobados
si no sumar 1 a reprobados
Sumar 1 a cant_alumnos
Imprimir "Total de aprobados="
Imprimir aprobados
Imprimir "Total de reprobados="
Imprimir reprobados

7. Programas ejemplo

7.1. Primer Programa en C

Imprimiremos la linea de texto: "Bienvenido a C!"

Pseudocodigo
Imprimir "Bienvenido a C!"

Programa
/* Primer programa en C */

```
main ()
{
      printf("Bienvenido a C!\n");
      system("PAUSE");
}
```

Veamos las lineas del programa:

```
/* Primer programa en C */
```

es un comentario. Escribimos comentarios colocando /* al principio y */ al final. Colocamos comentarios para facilitar el entendimiento del programa. Cuando se ejecuta el programa los comentarios se ignoran estos son ignorados por el compilador C y no generan código. Los comentarios ayudan a otras personas a leer y comprender el programa.

La linea

```
main ()
```

forma parte de todo programa de C. Los paréntesis despues de main indican que es una función. Los programas en C contienen una o más funciones una de las cuales deberá ser main. Todos los programas en C empiezan a ejecutarse en main.

La llave izquierda { debe de iniciar el cuerpo de cada función. Una llave derecha correspondiente debe de dar por terminada cada función.

La linea:

```
printf("Bienvenido a C!\n");
```

instruye a la computadora para que ejecute una acción, en este caso que imprima en la pantalla la cadena de caracteres entre comillas.

printf es una instrucción, toda instrucción debe terminar con un punto y coma.

La instrucción printf es una instrucción de biblioteca no forma parte del lenguaje. Cuando se compila y enlaza un programa se localizan las funciones de la biblioteca y se insertan las llamadas a estas funciones dentro del programa objeto.

Veamos aspectos de printf:

Cada llamada a printf contiene una cadena de control de formato que describe el formato de la salida. En este ejemplo en particular la cadena consiste en el string (secuencia de caracteres) "Bienvenido a C! \n". El \n final indica que se realize un salto de linea es decir despues de imprimir "Bienvenido a C!"se salta al comienzo de la linea siguiente.

Por último la instrucción:

```
system("PAUSE");
```

indica que el programa desplegue el mensaje:

Presione una tecla para continuar . . .

y que espere la entrada de una tecla cualquiera.
Para utilizar la función system uno debe incluir la biblioteca iostream.h.

7.2. Segundo Programa en C

Imprimiremos un valor aproximado de PI.

Pseudocodigo
Definir constante con valor de PI
Imprimir constante

Programa

```
/* Imprime valor aproximado de PI en la pantalla */

#include <stdio.h>
#include <iostream.h>

#define PI 3.1415926

main()
{
    printf("El valor aproximado de PI es: %f\n", PI);
    system("PAUSE");
}
```

Utilizamos printf y system, luego incluimos las bibliotecas correspondientes mediante las directrices:

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

La directiva :

```
#define PI 3.1415926
```

define una constante simbólica PI cuyo valor es 3.1415926. Una constante simbólica es un identificador que se reemplaza con texto de reemplazo en el preprocesador de C antes de que el programa sea compilado.

Observar que PI es un número real. En este caso definimos una constante con su valor.

Veamos la instrucción **printf**:

la cadena de control de formato contiene texto "El valor aproximado de PI es", contiene %f y \n.

El texto se imprime tal cual se colocó en la instrucción. %f es un especificador de conversión. Indica que se va a imprimir un número en punto flotante. \n indica que luego de imprimir el número se avanze a la linea siguiente.

El número en punto flotante se toma de la constante PI.

Los valores impresos con %f siempre imprimen por lo menos un digito a la izquierda del punto decimal. Además se colocan 6 digitos de presición a la derecha del punto decimal.

En este ejemplo se imprime:

El valor aproximado de PI es: 3.141693

Cuando imprimimos números en punto flotante podemos utilizar otros especificadores de conversión, en particular %e, %E (para imprimir float y double). %e imprime la e en miniscula. %E imprime la E mayuscula. Igual que %f imprimen un dígito a la izquierda del punto decimal. %e y %E muestran los números en notación exponencial.

7.3. Tercer Programa en C

Leeremos valores enteros ingresados en el teclado, calcularemos la suma e imprimiremos el resultado.

Pseudocodigo
Leer valor de x
Leer valor de y
Calcular suma=x+y
Imprimir suma

Programa

```
/* Suma dos numeros enteros */
#include <stdio.h>
#include <iostream.h>

main()
{
      int x,y,suma;
      printf("Ingrese 1er entero\n");
      scanf(" %d",&x);
      printf("Ingrese 2do entero\n");
      scanf(" %d",&y);
      suma=x+y;
      printf("La suma es %d\n",suma);
      system("PAUSE");
}
```

La linea:

int x,y,suma;

es una declaración. x,y y suma son nombres de variables. int es un tipo de datos.

Las variables son posiciones de memoria en las que se almacenan valores. Se están declarando tres variables que contendrán valores enteros.

Todas las variables deben de declararse con un nombre y un tipo de datos. En este caso declaramos las tres variables juntas, podriamos haber puesto:

```
int x;
int y;
int suma;
```

Observar que las declaraciones terminan en ;. Las declaraciones de variables se pueden realizar en cualquier punto de un programa (debe ser antes de utilizarlas).

Los nombres de variable deben ser identificadores válidos. Un identificador es una serie de caracteres formados de letras, digitos y subrayados (_), que no se inicien con un digito.

Mayusculas y minusculas se consideran diferentes.

Veamos la primer instrucción **scanf**:

```
scanf(" %d",&x);
```

toma la entrada del teclado. Tiene como argumentos %d y &x. El primer argumento es la cadena de control que indica de que tipo de datos es el valor que debe ingresar el usuario. %d indica debe ser un entero. El segundo argumento indica en que variable se debe almacenar el número leido, en este caso x.

Debe colocarse & antes de la variable excepto cuando se está leyendo una cadena.

Podriamos leer x e y juntos con la siguiente instrucción: (en general podemos leer cualquier cantidad de variables)

```
scanf(" %d %d",&x, &y);
```

se deben ingresar la x y la y separados por espacio, nueva linea o tabulador.

La instrucción:

```
suma=x+y;
```

calcula la suma de x e y y coloca el resultado en la variable suma.

La instrucción printf

```
printf("La suma es %d\n",suma);
```

declara la suma a imprimir es un entero (con %d). Podriamos haber realizado la suma en la instrucción printf escribiendola del siguiente modo:

```
printf("La suma es %d\n",x+y);
```

en cuyo caso no es necesario declarar la variable suma.

Principios de Programación

El lenguaje C

1. Identificadores, constantes y variables

1.1. Conceptos de memoria

Los nombres de variable como x, y, suma corresponden a localizaciones o posiciones en la memoria de la computadora.

Cada variable tiene un nombre, un tipo y un valor.

Como ya vimos, tenemos restricciones en nombres de variables y constantes. Deben estar formados por letras o digitos o guión bajo (_) y el primer caracter debe ser una letra. Letras pueden ser minusculas o mayusculas y estas se consideran diferentes.

Hay palabras reservadas como por ejemplo: **if, else, int, float**, etc.

1.2. Declaración de constantes

Podemos declarar constantes de dos modos diferentes: con #define que asocia un valor a un identificador, o con la instrucción:

const float PI 3.1415926;

que declara que PI es una variable de tipo punto flotante y que es constante (no se puede modificar, cualquier intento de modificarla da error).

En los programas que vimos antes, cuando se ejecuta la sentencia:

scanf(" %d", &x);

el valor escrito por el usuario se coloca en la posición de memoria a la cual se le ha asignado el nombre x.

Si por ejemplo el usuario ingresa el valor 15 este se colocará en la posición de x.

Siempre que se coloca un valor en una posición de memoria, este valor sustituye cuaquier valor anterior que tuviera la variable.

1.3. Tipos de datos simples

Tenemos los siguientes tipos de datos en C con sus largos respectivos:

char	1 byte
bool	1 byte
int	2 bytes
short	2 bytes
long	4 bytes
unsigned	4 bytes
float	4 bytes
double	8 bytes

char almacena caracteres. bool es el tipo de los valores de verdad que son true y false. Podemos colocar en vez de true 1 y en vez de false 0.

las variables int pueden ser calificadas como short, long o unsigned. short y long se refieren a distintos tamaños, unsigned son siempre positivos.

Las declaraciones de estos enteros son:

```
short int x;
long int y;
unsigned int z;
```

la palabra int se puede omitir.

El tamaño de los tipos depende del sistema operativo.

1.4. Valores constantes

Para enteros se utiliza el número correspondiente, por ejemplo: 123, 5, -8. Una constante entera que es muy grande para entrar en un int se toma como long.

Para reales, se puede utilizar la notación científica : 123.45e-7 o 0.12E3. Las constantes punto flotante se almacenan como double.

Una constante de caracter se coloca entre comillas simples: 'a', 'B', '\ n'. El valor de una constante de caracter es el valor numérico del caracter en la máquina. Por ejemplo en la codificación ASCII el 0, '0' es 48. Por ejemplo, supongamos tengo el siguiente programa:

```
main ()
{
      char c='a';
      printf("El valor de c es: %c\n", c);
      printf("El valor numerico de c es: %d\n", c);
      system("PAUSE");
}
```

imprimira:

```
El valor de c es: a
El valor de c es: 97
```

Las constantes de caracter, representan también caracteres como ser, salto de linea, tabulares

1.5. Declaraciones

Todas las variables deben ser declaradas antes de su uso. Una declaración especifica un tipo y es seguida por uno o más nombres de variables.

Podemos inicializar las variables en su declaración, como por ejemplo en:

```
int x=5;
int y=7;
long f1=0,f2=3.5;
```

2. Asignación y aritmética en C

2.1. Asignación

Una asignacion en C es una instrucción de la forma var=e donde var es una variable y e es una expresión del mismo tipo que var.

Ejemplos:

- Supongamos declaramos int x,y; podemos realizar la asignación: x=2+y.
- Supongamos declaramos bool b1,b2; podemos realizar la asignación b1=!b2 donde ! representa el not.
- Supongamos declaramos float f1=9.3,g1=3; podemos realizar la asignación f1=g1/f1;
- La asignación es una expresión y varias asignaciones se asocian de derecha a izquierda. Puedo escribir:

  x=y=z+2

  esto lo que hace es: calcula z+2, lo asigna a y y luego asigna y a x. No podemos escribir por ejemplo x=y+1=z;

2.2. Operaciones aritméticas

La mayor parte de los programas C ejecutan cálculos aritméticos. Los operadores aritméticos de C son:

Operación	Operador	Ejemplo
Suma	+	f+7
Substracción	-	p-c
Multiplicación	*	b*m
División	/	x/y
Módulo	%	r

Utilizamos símbolos especiales, no necesariamente iguales a los que utilizamos en matemática.

El asterisco (*) indica multiplicación y el signo de porcentaje (%) denota módulo (resto de la división).

Los operadores aritméticos son operadores binarios (se aplican a dos argumentos), excepto el menos (-) unario que calcula el opuesto.

Estos operadores están *superpuestos* es decir tienen significado cuando se aplican a distintos tipos de datos como ser enteros y reales excepto módulo que tiene significado solo cuando se aplica a enteros.

En particular la división cuando se aplica a enteros *trunca* es decir redondea a un entero (la división de enteros da como resultado un entero).

ejemplos:

7/4 = 1
7,0/4 = 1,75
7,0/4,0 = 1,75
7 %4 = 3
17 %5 = 2

Un error común en programación es la división por 0.

Los paréntesis se utilizan del mismo modo qu en matemáticas. Por ejemplo para multiplicar a por (b+c) escribimos $a * (b + c)$.

C calculará las expresiones aritméticas en una secuencia precisa determinada por las reglas de precedencia de operadores que siguen y que en general son las mismas que en matemáticas, a saber:

1. Primero se calculan las expresiones contenidas en paréntesis. En el caso de paréntesis anidados se evalua primero la expresión en el par de paréntesis más interno.

2. A continuación se evalúan las operaciones de multiplicación división y módulo. Si hay varias se evalua primero de izquierda a derecha. Se dice que estas operaciones tienen el mismo nivel de precedencia.

3. Por último se calculan las operaciones de suma y resta. Igual que en el caso anterior si hay varias de estas operaciones se asocia de izquierda a derecha y se dice que tienen el mismo nivel de precedencia.

Ejemplo

- x= a*(b+c)/d+e*5

  primero evaluamos la expresión entre parentesis: (b+c) a continuación a* el resultado de la suma anterior a continuación dividimos por d luego calculamos e*5 y por ultimo sumamos el resultado de a*(b+c)/d a e*5.

  Podemos para simplificar agregar parentesis y luego evaluar. Obtenemos la siguiente expresión.

  ((a*(b+c))/d)+(e*5)

- z=p*r %q+w/x-y

  agreguemos parentesis, obtenemos:

  (((p*r) %q)+(w/x))-y

2.3. Operadores de igualdad y relacionales

Las expresiones booleanas (expresiones que tienen como valor true o false) se forman utilizando los operadores de igualdad y los operadores relacionales.

Los operadores relacionales tienen el mismo nivel de precedencia y se asocian de izquierda a derecha.

Los operadores de igualdad tienen un nivel de precedencia menor que los relacionales y también se asocian de izquierda a derecha.

Los operadores son los siguientes:

Operador de igualdad	significa	ejemplo de condición	es verdadero si
==	igualdad	x==y	x es igual a y
!=	distinto	x!=y	x es distinto a y

Operador relacional	significa	ejemplo de condición	es verdadero si
>	mayor	x>y	x es mayor que y
<	menor	x<y	x es menor que y
>=	mayor o igual	x>=y	x es mayor o igual a y
<=	menor o igual	x<=y	x es menor o igual a y

2.4. Operadores booleanos

Los operadores booleanos o conectivos lógicos son:

Operador	Significado	ejemplo	verdadero si
&&	AND	x && y	x e y son verdaderos, falso en otro caso
||	OR	x || y	verdadero si x o y son verdaderos, falso en otro caso
!	NOT	!x	verdadero si x es falso, falso si x es verdadero

2.5. Operadores de asignación

C dispone de operadores de asignación que *abrevian* las expresiones de asignación. Por ejemplo,

c=c+3;

se puede escribir en la forma

c+=3;

En general,

var = var op expr

donde $op \in \{+, -, *, /, \&\}$ puede ser escrito en la forma

var op = expr

Por ejemplo:

Operador	ejemplo	equivalente a
+=	c +=7	c=c+7
-=	d -=4	d=d-4
=	e*=5	e=e*5
/=	f/=3	f=f/3
%=	g %=9	g=g %9

2.6. Operadores incrementales y decrementales

C dispone de operadores incremental unario (++) y decremental unario (–). Podemos utilizar c++ en vez de c=c+1 o c+=1.

Si los operadores son colocados antes de la variable (++c,–c) se conocen como operadores de preincremento y predecremento, si se colocan después (c++,c–) se conocen como operadores de postincremento y postdecremento.

El preincrementar o decrementar una variable hace que la variable primero se incremente o decremente en 1 y a continuación el nuevo valor de la variable se utilizará en la expresión en la cual aparece.

por ejemplo:

```
int c;

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

el primer printf incrementa c, luego se imprime 6. El segundo printf imprime 6 y el tercero 5.

El postincrementar o decrementar una variable hace que el valor de la variable primero se utilize en la expresión en la cual aparece y luego se incremente o decremente en 1.

por ejemplo:

```
int c;

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

el primer printf imprime c (5) y luego lo incrementa. El segundo printf imprime 6 y el tercero 6 y luego decrementa. El cuarto printf imprime 5.

2.7. Operador condicional

Es el único operador ternario de C (utiliza tres operandos). Los operandos junto con el operador condicional conforman una expresión condicional. Por ejemplo:

grado >= 60 ? printf("Aprobado\n") : printf ("Reprobado\n");

La idea es : se evalua la primer expresión que debe dar como resultado un booleano. Si la expresión evalua a verdadero se evalua la segunda expresión (la que está a la derecha de ?), si la expresión evalua a falso se evalua la tercer expresión (la que está a la derecha de ':').

2.8. Precedencia y Asociatividad

La precedencia y asociatividad de los operadores vistos hasta ahora es:

Operador	asociatividad	tipo
()	izq a der	paréntesis
++, –, -, !	der a izq	unario
,/, %	izq a der	multiplicativo
+,-	izq a der	aditivo
¡,¡=,¿,¿=	izq a der	relacional
==,!=	izq a der	igualdad
&&	izq a der	AND lógico
——	izq a der	OR lógico
?:	der a izq	condicional
=, +=, -=, *=, /=, %=	der a izq	asignación

Ejemplos de precedencia

- x==y+1>=z-3

  coloquemos paréntesis: (x==((y+1)>=(z-3)))

- x=y+1!=z+3

  coloquemos paréntesis:

  x=((y+1)!=(z+3))

Principios de Programación

El lenguaje C

1. Ejemplos correspondientes al teorico2

1.1. Conceptos de memoria, nombres de variables

Ejemplos de nombres de variable validos son:

i, i1, i2, int1, int2, f, floa, double1

ejemplos de nombres de variable no válidos son:

2i, 4fl, int, float, double, if

los primeros son no válidos por empezar con números, los siguientes son palabras reservadas.

1.2. Declaración de constantes

ejemplo:

const float PI 3.1415926;

en general se utiliza:

const tipo nombre_de_variable valor;

lo que hacemos es declarar una variable inicializada con un valor. El declararla como const (poner const antes del tipo) es lo que hace que sea una constante. Las constantes no se pueden modificar (es un error de sintaxis).

1.3. Operaciones aritméticas

Ejemplos de usos de operaciones aritméticas en programas C son:

```
x=x*2+1;
x=y*3-5+z;
z=(x-1)/2;
z=(x%2)*y;
```

donde utilizamos las operaciones dentro de expresiones que se encuentran en la operación de asignación.

1.4. Operadores de igualdad y relacionales

Ejemplos de operadores de igualdad y relacionales son los siguientes:

```
x+2 == y-1
(z*3)+1 != (y+2)*3
x>y
x+2 <= y+3
x-2 >= (y-1)*2
```

es usual utilizar expresiones de igualdad o relacionales en instrucciones especificas que contienen condiciones lógicas.

Si pensamos en pseudocódigo, las instrucciones si, si/si no y mientras contenian expresiones lógicas. Las intrucciones correspondientes en C tambien tienen condiciones lógicas y la escribimos como en los ejemplos de arriba.

1.5. Operadores booleanos

Ejemplos de operadores booleanos son condiciones lógicas como las anteriores conectadas por and, or y not. Por ejemplo:

```
(x > y && y >= z)
(x+2 == z && z+3<y)
(x || y); /* x e y son variables booleanas */
(x+3 < y%2 || !(z==y))
```

1.6. Operadores de asignación

Ejemplos de operadores de asignación:

```
c+=3*z; /* corresponde a c=c+3*z; */
d*=2-y+x; /* corresponde a d=d*(2-y+x);*/
e%=2; /* corresponde a e=e%2;*/
```

1.7. Operadores incrementales y decrementales

Podemos utilizarlos en el lugar de cualquier expresión aritmética, por ejemplo:

```
x=c++; /* asignacion */
y=x+(z++); /*asignacion */
x+2 == −− y; /* expresion booleana */
y*5 ! = ++z + −−x; /* expresion booleana */
y-3 > ++x; /* expresion relacional */
y+x <= z −− * 5; /* expresion relacional */
```

1.8. Operador condicional

Ejemplos:

```
x>0 ? 5 : 3; /* retorna un valor */
x+y <= z*6 ? x=x+2 : x=x+1; /* ejecuta una instruccion */
```

Principios de Programación

El lenguaje C

1. Inicialización de variables

Podemos inicializar una variable cuando la declaramos poniendo nombre = valor o por una instrucción de asignación dentro del programa o mediante una lectura que asigna un valor a la variable.

Si no se inicializa una variable su valor es indefinido. Si por ejemplo imprimieramos su valor sin haberla inicializado se imprimiria cualquier valor.

2. Conversión entre tipos de variables

Hay casos en los cuales se realiza una conversión automática de los valores de variables de distintos tipos.

Por ejemplo, supongamos tenemos las siguientes declaraciones:

```
int i1=1,i2=2;
float f1=1.1, f2=2.2;
```

si realizamos la siguiente operación:

```
i1=f1;
```

se trunca el valor de f1, se toma su parte entera y se asigna a i1.

si realizamos la siguiente operación:

```
f1=i2;
```

se asigna el valor de i2 a f1 sin conversión.

si realizamos la siguiente operación:

```
f1=i1+f1;
```

se realiza la suma del mismo modo que si i1 fuera punto flotante.

```
f2 = i2/i1;
```

i2/i1 es una división entera, luego se asigna el valor entero a f2.

```
f2 = i2/f1;
```

es una división real pues lo es f1. Se asocia el real correspondiente a f2.

2.1. Cast

Si en el programa anterior quisieramos que la operación

f2=i1/i2;

diera un valor real, podemos utilizar casting que es una conversión explicita de un tipo a otro. Tendriamos que escribir la operación en la forma:

f2=(float) i1/i2;

(float) indica se realize un cálculo de punto flotante. Lo que realiza este operador es crear una copia temporal de punto flotante de su operando en este caso i1. El uso de un operador cast de esta forma se conoce como *conversión explicita*. El valor almacenado en i1 continua siendo un entero. El cálculo ahora consiste en un valor de punto flotante (la versión temporal de i1) dividida por el valor entero almacenado en i2. El compilador de C solo sabe como evaluar expresiones en donde los tipos de datos de los operandos sean identicos. A fin de asegurarse que los operandos sean del mismo tipo el compilador lleva a cabo una operación denominada *promoción* (también conocida como *conversión implicita*) sobre los operandos. Por ejemplo en una expresión que contenga los datos int y float se hacen copias de los operandos int y se promueven a float.

Los operadores cast están disponibles para cualquier tipo de datos. El operador cast se forma colocando parentesis alrededor del nombre de un tipo de datos. El operador cast es un *operador unario* (es decir que utiliza un operando). Los operadores cast se asocian de derecha a izquierda y tienen la misma precedencia que los otros operadores unarios.

Principios de Programación

El lenguaje C

1. Instrucciones de control

1.1. Secuencia, selección, iteración

Por lo regular en un programa los enunciados son ejecutados uno después del otro, en el orden en que aparecen escritos. Esto se conoce como secuencia.

C proporciona tres tipos de estructura de selección: la estructura **if** que elige una acción si una condición es verdadera o pasa por alto la acción si la condición es falsa, la estructura **if/else** que ejecuta una acción si la condición es verdadera y otra acción diferente si la condición es falsa y la estructura de selección **switch** que veremos más adelante, que ejecuta una entre muchas acciones diferentes dependiendo del valor de una expresión.

La estructura **if** se llama *estructura de una sola selección*, **if/else** se conoce como *estructura de doble selección* y la estrutura **switch** se conoce como *estructura de selección múltiple.*

C proporciona 3 tipos de estructura de repetición: **while** que significa mientras, **do/while** que significa hacer mientras y **for** que significa para. Veremos la primera y más adelante las otras.

1.2. La estructura de selección if

Consideremos el pseudocódigo:

Si la nota del estudiante es mayor o igual a 60
  Imprimir "Aprobado"

Lo codificamos en C del siguiente modo:

```
if (nota >= 60) printf("Aprobado");
```

En general una sentencia if consiste de :

1. La palabra clave **if**
2. Una condición booleana entre parentesis
3. Una instrucción que se ejecutará si la condición es verdadera.

En el ejemplo anterior la condición es (nota >= 60) y la instrucción es printf("Aprobado");

1.3. La estructura de selección if/else

Consideremos el pseudocódigo:

```
Si la nota del estudiante es mayor o igual a 60
   Imprimir "Aprobado"
si no Imprimir "Reprobado"
```

Lo codificamos en C del siguiente modo:

```
if (nota >= 60) printf("Aprobado");
else printf("Reprobado");
```

En general una sentencia if/else consiste de :

1. La palabra clave **if**
2. Una condición booleana entre parentesis
3. Una instrucción que se ejecutará si la condición es verdadera.
4. la palabra clave **else**
5. Una instrucción a ejecutar si la condición es falsa.

En el ejemplo anterior la condición es (nota >= 60), la primer instrucción es printf("Aprobado"); y la segunda instrucción es printf("Reprobado");.

El operador condicional (?:) está relacionado de cerca con la estructura **if/else**.

Este operador se escribe en la forma:

```
a ? b : c;
```

donde a es una expresión booleana y b y c pueden ser expresiones o sentencias. Si el valor de a es verdadero se devuelve el valor de b y si el valor de a es falso se devuelve el valor de c.

Podemos escribir en vez de la instrucción anterior la siguiente:

```
printf("%s\n", nota >= 60 ? "Aprobado":"Reprobado");
```

"$\%s$" indica que se imprime una cadena de caracteres.

También podemos escribir la instrucción del siguiente modo: (los valores de una expresión condicional pueden ser acciones a ejecutar).

```
nota >= 60 ? printf("Aprobado"):printf("Reprobado");
```

Podemos anidar estructuras if e if/else. Esto ocurre cuando una de las sentencias a ejecutar en un if es otro if. Por ejemplo:

```
if (nota >= 90) printf("la nota es A");
else
    if (nota >= 80) printf("la nota es B");
    else
        if (nota >= 70) printf("la nota es C");
        else
            if (nota >= 60) printf("la nota es D");
            else printf("Reprobado");
```

La estructura if espera un enunciado dentro de su cuerpo. Si el enunciado está compuesto por otros varios enunciados los colocamos entre llaves { y }. Un conjunto de instrucciones contenidos entre llaves se conoce como *enunciado compuesto.*

Por ejemplo:

```
if (grado >= 60) printf("Aprobado");
else
    {
      printf("Reprobado\n");
      printf("Debe repetir el curso");
    }
```

Si no colocaramos las llaves, el segundo printf del else se consideraría como instrucción siguiente al if/else y se ejecutaria siempre.

Otro caso de anidamiento:

supongamos tenemos el siguiente programa:

```
if (n>0)
  if (i<j)
  {
    printf ("Valor de i %d", i);
    printf ("Valor de j %d", j);
  }
else
  printf ("error 1");
```

el else está indentado de modo que parecería que corresponde al 1er if, sin embargo el compilador lo va a asociar al segundo if. Si quisieramos que se corrispondiera con el primer if debemos poner el segundo if dentro de llaves como a continuación:

```
if (n>0)
  {
   if (i<j)
   {
    printf ("Valor de i %d", i);
    printf ("Valor de j %d", j);
   }
  }
else
  printf ("error 1");
```

1.4. La estructura de repetición while

Una estructura de repetición le permite al programador especificar que se repita una acción mientras cierta condición se mantenga verdadera.

El enunciado en pseudocódigo:

```
Mientras queden elementos en mi lista de compras
        Adquirir elemento siguiente y tacharlo de la lista
```

describe la repetición que ocurre durante una salida de compras. El enunciado o enunciados contenidos en un while constituyen el *cuerpo del while*. Puede ser un enunciado sencillo o un enunciado compuesto entre llaves.

En el ejemplo anterior y en general cuando escribimos un while se espera que la condición alcance en algún momento el valor falso (si esto no ocurriera entrariamos en un loop infinito).

Por ejemplo, el programa que calcula la primer potencia de 2 superior a 1000 cuyo pseudocódigo es :

```
Definir producto=2.
Mientras producto <= 1000
        multiplicar producto por 2.
Imprimir producto.
```

podemos escribir el programa correspondiente como sigue:

```
producto=2;
while (producto <= 1000)
      producto = 2 * producto;
printf("El primer numero potencia de 2 superior a 1000 es %d", producto);
```

2. Ejemplos de programas simples con while

2.1. Factorial

Pseudocódigo

```
Imprimir solicitud de ingreso de n.
Leo n.
Inicializo i con 1.
Inicializo fac con 1.
Mientras i sea menor o igual a n
        Calcular fac=fac*i.
        Sumar 1 a i.
Imprimir fac.
```

Programa C

```
main()
{
     int i,n,fac;
     printf("Ingrese numero del cual calculara factorial \n");
     scanf( %d,&n);
     i=1;
```

```
    fac=1;
    while (i<=n)
    {
     fac*=i;
     i++;
    }
    printf("Factorial de %d es igual a %d",n,fac);
    system("PAUSE");
}
```

2.2. Potencia n-esima de un número dado

Pseudocódigo

```
Imprimir solicitud de ingreso de n.
Imprimir solicitud de ingreso de numero.
Leo n.
Leo num.
Inicializo i con 1.
Inicializo potencia con 1.
Mientras i sea menor o igual a n
        Calcular potencia=potencia * num.
        Sumar 1 a i.
Imprimir potencia.
```

Programa C

```
main()
{
    int i,n,num,potencia;
    printf("Ingrese potencia \n");
    scanf("%d,&n);
    printf("Ingrese numero cuya potencia desea calcular\"n);
    scanf("%d,&num);
    i=1;
    potencia=1;
    while (i<=n)
    {
     potencia=potencia*num;
     i++;
    }
    printf("Potencia %d de %d es igual a %d",n,num,potencia);
    system("PAUSE");
}
```

A continuación veremos los programas correspondientes a los ejemplos de repetición vistos en un teórico anterior.

3. Casos de Estudio de programación con while

3.1. Repetición controlada por contador

Consideremos el enunciado:

Una clase de diez alumnos hizo un examen. Las calificaciones (enteros en el rango de 0 a 100) correspondientes a este examen están a su disposición. Determine el promedio de la clase en este examen.

cuyo pseudocódigo es:

```
Inicializar total en 0
Inicializar nro_alumno a 1
Mientras nro_alumno menor o igual a 10
        Leer siguiente calificacion
        Sumar calificacion a total
        Sumar 1 a nro_alumno
Calcular promedio = total /10
Imprimir promedio
```

el programa correspondiente es:

```
main ()
{
    total=0;
    nro_alumno=1;
    while (nro_alumno <= 10)
    {
     scanf("%d",&calificacion);
     total+=calificacion;
     nro_alumno++;
    }
    promedio=total/10;
    printf("El promedio es %f",promedio);
    system("PAUSE");
}
```

3.2. Repetición controlada por centinela

El problema es igual al anterior salvo que en vez de considerar 10 estudiantes consideramos una cantidad arbitraria.

Ingresaremos datos y cuando hayamos terminado con todos los datos ingresaremos -1.

El pseudocódigo es el siguiente:

```
Inicializar total en 0
Inicializar cant_alumnos a 0
Leer centinela
Mientras centinela distinto de -1
```

```
        Leer siguiente calificacion
        Sumar calificacion a total
        Sumar 1 a cant_alumnos
        Leer centinela
Si cant_alumnos es distinto de 0 entonces
        calcular promedio = total /cant_alumnos
        Imprimir promedio
si no Imprimir "no se ingresaron calificaciones"
```

El programa correpondiente es:

```
main () {
    total=0;
    cant_alumnos=0;
    scanf(" %d",&centinela);
    while (centinela!=-1)
    {
     scanf(" %d", &calificacion);
     total+=calificacion;
     cant_alumnos++;
     scanf(" %d", &centinela);
    }
    if (cant_alumnos!=0)
    {
     promedio=total/cant_alumnos;
     printf("El promedio es %f", promedio);
    }
    else
     printf("No se ingresaron calificaciones");
    system("PAUSE");
```

3.3. Estructuras de control anidadas

Consideremos el problema:

Una universidad ofrece un curso que prepara alumnos para un examen. La universidad desea saber que tan bien salieron sus alumnos en el examen. Se ingresará un 1 si el alumno pasó el examen y un 2 si lo reprobó. Se quiere saber total de aprobados y total de reprobados en un total de 10 alumnos.

Cuyo pseudocódigo es:

```
Inicializar aprobados en 0
Inicializar reprobados en 0
Inicializar cant_alumnos a 1
Mientras cant_alumnos menor o igual a 10
        Leer siguiente calificacion
        Si calificacion = 1 sumar 1 a aprobados
        si no sumar 1 a reprobados
```

Sumar 1 a cant_alumnos
Imprimir “Total de aprobados=”
Imprimir aprobados
Imprimir “Total de reprobados=”
Imprimir reprobados

el programa correspondiente es:

```
main ()
{
      total=0;
      aprobados=0;
      reprobados=0;
      cant_alumnos=1;
      while (cant_alumnos <= 10)
      {
       scanf(" %d", &calificacion);
       if (calificacion==1) aprobados++;
       else reprobados++;
       cant_alumnos++;
      }
      printf("Total de aprobados");
      printf(" %d\n", aprobados);
      printf("Total de reprobados");
      printf(" %d", reprobados);
      system("PAUSE");
```

4. Una colección de programas utiles

4.1. Entrada y salida de caracteres

La biblioteca estandar provee funciones para leer y escribir caracteres. La función getchar() ingresa el próximo caracter del teclado cada vez que es invocada (retorna el caracter como valor). Por ejemplo:

```
c=getchar()
```

coloca el caracter leido en c.

La función putchar(a) es el complemento de getchar. Imprime en pantalla el valor de la variable a.

Podemos mezclar printf, scanf, getchar y putchar.

4.2. Copia de entrada en la salida

Pseudocódigo

Leer un caracter.
Mientras no sea final de archivo
        Imprimir el caracter.
        Leer el siguiente caracter.

Hay un valor especial que indica fin de archivo que es el -1 o CTRL-Z (EOF). El programa correspondiente al pseudocódigo anterior es:

Programa

```
main()
{
        int c;

        c=getchar();
        while(c!=EOF)
        {
                putchar(c);
                c=getchar();
        }
}
```

Podemos escribir el programa en la siguiente forma: (asociamos la condición del while con el getchar(), cuando se ejecuta el programa se lee el caracter y se devuelve como valor de la expresión el caracter leido)

Programa

```
main()
{
        int c;

        while((c=getchar())!=EOF)
                putchar(c);
}
```

getchar() ingresa blancos, nueva linea y tabuladores, (si ingreso este tipo de caracter, el putchar lo imprime).

4.3. Cuenta de la cantidad de caracteres leidos

Pseudocódigo

Contador=0.
Leer un caracter.
Mientras no sea final de archivo
        sumar 1 a contador.
Imprimir cantidad de caracteres leidos.

Programa

```
main()
{
      long nc;

      nc=0;
      while (getchar()!=EOF) nc++;
      printf(" %ld\n", nc);
      system("PAUSE");
}
```

el %ld en printf indica se imprime un long integer.

4.4. Cuenta de la cantidad de lineas leidas

Pseudocódigo

```
Contador=0.
Leer un caracter.
Mientras no sea final de archivo
          si el caracter leido es "\n"sumar 1 a contador.
Imprimir cantidad de lineas leidas.
```

Programa

```
main()
{
      int c,nl;

      nl=0;
      while ((c=getchar())!=EOF)
            if (c=='\n') nl++;
      printf(" %d\n", nl);
      system("PAUSE");
}
```

4.5. Cantidad de nueva linea, palabras y caracteres

Pseudocódigo

```
setear en_palabra a falso
setear nl, nw y nc a cero
       comentario : nl cuenta cantidad de lineas, nw de palabras y nc de caractereres.
mientras el siguiente caracter no sea EOF
          sumar uno a nc.
          si el caracter es nueva linea sumar 1 a nl.
          si el caracter es blanco, tab o nueva linea
             setear en_palabra a falso.
```

```
        si no
          si en_palabra es falso
            setear en_palabra a verdadero.
            sumar 1 a nw.
Imprimir nl, nw, nc.
```

Cada vez que el programa encuentra el primer caracter de una palabra lo cuenta. La variable en_palabra recuerda cuando el programa está dentro de una palabra y cuando no. Inicialmente no lo es. Cuando no está dentro de una palabra y el siguiente caracter no es espacio, nueva linea ni tabulador seteamos que está dentro de una palabra. Se suma uno a cantidad de palabras.

Programa

```
main()
{
    int c, nl, nw, nc, en_palabra;

    en_palabra = false;
    nl=nw=nc=0;
    while ((c=getchar())!=EOF)
    {
        ++nc;
        if (c=='\n') ++nl;
        if (c==' ' || c=='\n' || c=='\t') en_palabra=false;
        else if (!en_palabra) { en_palabra=true; ++nw; }
    }
    printf(" %d %d %d \n", nl, nw, nc);
    system("PAUSE");
}
```

Principios de Programación

El lenguaje C

1. Más sobre Instrucciones de control

Estudiaremos con mayor detalle la repetición y presentaremos estructuras adicionales de control de la repetición a saber las estructuras **for** y **do-while**.

Presentaremos la estructura de selección múltiple **switch**.

Analizaremos los enunciados **break** que sale de las estructuras de control y **continue** que saltea el resto de una estructura de control y continua con la siguiente iteración.

1.1. Lo esencial de la repetición

La mayor parte de los programas incluyen repeticiones o ciclos. Un ciclo es un grupo de instrucciones que la computadora ejecuta en forma repetida en tanto se conserve verdadera alguna condición de continuación del ciclo. Hemos analizado dos procedimientos de repetición:

1. repetición controlada por contador
2. repetición controlada por centinela

La repetición controlada por contador se denomina a veces *repetición definida* porque con anticipación se sabe con exactitud cuantas veces se ejecutará el ciclo. La repetición controlada por centinela a veces se denomina *repetición indefinida* porque no se sabe con anticipación cuantas veces se ejecutará el ciclo.

En la repetición controlada por contador se utiliza una *variable de control* para contar el número de repeticiones. La variable de control es incrementada (normalmente en 1) cada vez que se ejecuta el grupo de instrucciones dentro de la repetición. Cuando el valor de la variable de control indica que se ha ejecutado el número correcto de repeticiones se termina el ciclo y la computadora continúa ejecutando el enunciado siguiente al de la estructura de repetición.

Los valores centinela se utilizan para controlar la repetición cuando:

1. El número preciso de repeticiones no es conocido con anticipación, y
2. El ciclo incluye enunciados que deben obtener datos cada vez que este se ejecuta.

el valor centinela indica "fin de datos". El centinela es introducido una vez que al programa se le han proporcionado todos los datos. Los centinelas deben ser un valor diferente a los valores que toman los datos.

1.2. Repetición controlada por contador

La repetición controlada por contador requiere:

1. El nombre de una variable de control (o contador del ciclo).
2. El valor inicial de la variable de control.
3. El incremento (o decremento) con el cual, cada vez que se termine un ciclo la variable de control será modificada.
4. La condición que compruebe la existencia del valor final de la variable de control.

Considere el siguiente programa que imprime los números del 1 al 10:

Programa

```
#include <stdio.h>

main ()
{
      int contador=1;

      while (contador <= 10) {
            printf (" %d\n", contador);
            ++contador;
      }
      system("PAUSE");
}
```

La declaración contador=1 le da nombre a la variable de control, la declara como un entero, reserva espacio para la misma y pone su valor inicial a 1.

La declaración e inicialización de contador también podría haber sido hecha mediante los enunciados:

```
int contador;

contador=1;
```

El enunciado ++contador; incrementa en 1 el contador del ciclo cada vez que este se ejecuta. La condición del while prueba si el valor de la variable de control es menor o igual a 10 (un valor mayor en este caso 11) hace que la condición se vuelva falsa.

Alternativamente podriamos haber escrito:

```
contador=0;
while (++contador <= 10)
      printf(" %d\n", contador);
```

este trozo de programa incrementa el contador y testea la condición juntos en la condición del while.

2. La estructura de repetición for

La estructura de repetición **for** maneja de forma automática todos los detalles de la repetición controlada por contador.

Para ilustrar los poderes del for reescribiremos el programa anterior:

Programa

```
main ()
{
      int contador;
      for (contador=1; contador <= 10; contador++)
          printf(" %d\n, contador);
      system("PAUSE");
}
```

El programa funciona como sigue: cuando la estructura for inicia su ejecución la variable contador se inicializa en 1. A continuación se verifica la condición de continuación del ciclo que es contador ¡= 10. Como la condición se satisface se imprime contador y luego la variable contador se incrementa en 1, y el ciclo comienza de nuevo con la prueba de continuación del ciclo.

En general:

```
for (expresion1; expresion2; expresion3)
    enunciado;
```

es equivalente a:

```
expresion1;
while (expresion2)
{
      enunciado;
      expresion3;
}
```

expresión 1 y expresión 3 pueden ser cada una de ellas un conjunto de instrucciones separadas por comas que se evaluaran de izquierda a derecha. El valor y el tipo de una lista de expresiones separadas por coma es el valor y el tipo de la expresión de más a la derecha de la lista. El operador coma es utilizado prácticamente solo en estructuras for. Su uso principal es permitir al programador utilizar múltiples expresiones de inicialización y/o múltiples incrementos.

Las tres expresiones de la estructura for son opcionales. Si se omite expresion2, C supondrá la condición es verdadera creando un ciclo infinito. Se puede omitir expresion1 si la variable de control se inicializa en algún otro lado. La expresión3 podría también omitirse si el incremento no es necesario o si se lo calcula mediante enunciados en el cuerpo de la estructura for. Los dos puntos y coma de la estructura for son requeridos.

2.1. La estructura for: notas y observaciones

1. El incremento puede ser negativo (el ciclo ira hacia atrás).
2. Si la condición de continuación del ciclo resulta falsa al inicio, la porción del cuerpo del ciclo no se ejecutará. En vez de ello, la ejecución seguirá adelante con el enunciado que siga a la estructura for.
3. La variable de control, con frecuencia se imprime o se utiliza en cálculos en el cuerpo del ciclo pero esto no es necesario. Es común utilizar la variable de control para controlar la repetición sin necesidad de utilizarla dentro del cuerpo del ciclo.

2.2. Ejemplos utilizando la estructura for

Lo siguientes ejemplos muestran métodos de variar en una estructura for la variable de control:

1. Varie la variable de control de 1 a 100 en incrementos de a 1:

   for (i=1 ; i <= 100; i++)

2. Varie la variable de control de 100 a 1 en incrementos de -1:

   for (i=100 ; i >= 1; i- -)

3. Variar la variable de control de 7 a 77 en pasos de 7:

   for (i=7; i <= 77; i+=7)

4. Variar la variable de control de 20 a 2 en pasos de -2:

   for (i=20; i >= 2; i-=2)

5. Variar la variable de control a lo largo de la siguiente secuencia de valores: 2, 5, 8, 11, 14, 17, 20:

   for (i=2; i <= 20; i+=3)

6. variar la variable de control de acuerdo a la siguiente secuencia de valores: 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

   for (i=99; i >= 0; i-=11)

Los siguientes dos ejemplos proporcionan aplicaciones simples de la estructura for. El primer programa utiliza la estructura for para sumar todos los enteros pares desde 2 hasta 100:

Programa 1

```
#include <stdio.h>

main ()
{
      int suma=0, numero;
      for (numero=2; numero <= 100; numero+=2)
          suma += numero;
      printf("Suma es %d\n", suma);
      system("PAUSE");
}
```

Podriamos realizar la cuenta de suma y las inicializaciones dentro del encabezado del for del siguiente modo:

Programa 1

```
#include <stdio.h>

main ()
{
      int numero;
      for (int suma=0, numero=2; numero <= 100; suma += numero, numero+=2);
      printf("Suma es %d\n", suma);
      system("PAUSE");
}
```

El segundo programa calcula el interes compuesto. Considere el siguiente enunciado del problema:

Una persona invierte 1000$ en una cuenta de ahorros que reditúa un interes del 5 %. Suponiendo que todo el interes se queda en depósito dentro de la cuenta, calcule e imprima la cantidad de dinero en la cuenta al final de cada año, durante 10 años. Utilice la fórmula siguiente:

$a = p(1 + r)^n$

donde:
p es la cantidad invertida
r es la tasa anual de interes
n es el numero de años
a es la cantidad en depósito al final del año n

Este problema incluye un ciclo que ejecuta el cálculo indicado para cada uno de los 10 años en los cuales el dinero se queda en depósito. El programa es el siguiente:

Programa 2

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

main ()
{
      int anio;
      double cantidad, principal=1000, tasa=.05;

      printf("%4s%21s\n", "Anio, "Cantidad en deposito");

      for (anio=1;anio <= 10;anio++)
      {
          cantidad = principal * pow(1.0+tasa,anio);
          principal = cantidad;
          printf("%4d%21.2f \n", anio, cantidad);
      }
}
```

pow(x,y) calcula el valor de x elevado a la y. Toma dos argumentos de tipo double y devuelve un valor double. Incluimos la biblioteca math que contiene la definición de esta función. El argumento year es entero. El archivo math.h incluye información que le indica al compilador que convierta el valor de year a una representación temporal double antes de llamar a la función.

El especificador de conversión "%21.2f.[es] utilizado en el programa para imprimir el valor de la variable cantidad. El 21 especifica el ancho del campo en el cual el valor se imprimirá. El 2 especifica la presición (número de posiciones decimales). Si el número de caracteres desplegados es menor que el ancho del campo el valor será automáticamente alineado a la derecha. Si quisiera alinear a la izquierda se debe colocar un signo de menos entre el % y el ancho de campo. El signo de menos puede utilizarse también para alinear enteros y cadenas de caracteres.

3. La estructura de selección múltiple switch

Un programa puede contener una serie de desiciones en las cuales una variable o expresión se probará por separado contra cada uno de los valores constantes enteros que pueda asumir y se tomarán diferentes acciones. Para esta forma de toma de decisiones se proporciona la estructura de decisión múltiple switch.

La estructura switch está formada de una serie de etiquetas **case** y de un caso opcional **default**.

En el siguiente programa se leen calificaciones y se cuenta el total de cada calificación ingresada. Dentro del encabezado de un while se leerá el grado de aprobación (que es un caracter). La repetición termina cuando se ingresa EOF.

Programa

```
#include <stdio.h>

main ()
{
      int grado;
      int cant_a=0, cant_b=0, cant_c=0,
          cant_d=0, cant_e=0, cant_f=0;

      printf ("Ingrese la calificacion\n");
      printf ("Ingrese EOF para terminar el ingreso de datos\n");
      while (( grado=getchar())!=EOF)
      {
            switch (grado)
            {
             case 'A': case 'a': cant_a++;
                                 break;
             case 'B': case 'b': cant_b++;
                                 break;
             case 'C': case 'c': cant_c++;
                                 break;
             case 'D': case 'd': cant_d++;
                                 break;
             case 'E': case 'e': cant_e++;
                                 break;
             case 'F': case 'f': cant_f++;
                                 break;
             case '\n': case ' ': break;
             default :
                       printf("Ha ingresado un grado incorrecto, entre un nuevo grado");
                       break;
            }
      }
printf("\n Totales de cada grado son:\n");
printf("A: %d\n", cant_a);
printf("B: %d\n", cant_b);
printf("C: %d\n", cant_c);
printf("D: %d\n", cant_d);
printf("E: %d\n", cant_e);
printf("F: %d\n", cant_f);
}
```

Una de las caracteristicas de C es que los caracteres pueden ser almacenados en cualquier tipo de datos entero, ya que son representados como enteros de 1 byte. Podemos entonces tratar a un carácter como si fuera un entero o un caracter dependiendo de su uso.

Los caracteres pueden ser leidos utilizando scanf mediante el uso del especificador de conversión $\%c$.

La expresión de control del switch es la que se coloca entre paréntesis a continuación de la palabra clave switch.

En el ejemplo anterior el usuario introduce las calificaciones con el teclado. Cuando se oprime enter los caracteres son leidos por getchar() de a un caracter a la vez. Si el caracter introducido no es igual a EOF (CTRL-Z) se introduce en la estructura switch. En el ejemplo, grado es la expresión de control. El valor de esta expresión es comparado con cada una de las etiquetas case. Suponga el usuario ha escrito la letra C como calificación. La C se compara con cada uno de los case dentro del switch. Si ocurre una coincidencia, se ejecutarán los enunciados correspondientes a dicho case. En este caso se incrementa cant_c en 1, y luego se ejecuta break que hace que se salga del switch.

El enunciado break causa que el control del programa continue con el primer enunciado que sigue despues de la estructura switch. Si no se utilizara el enunciado break los cases siguientes al de la coincidencia se ejecutarian (por ejemplo ingreso solo letras a y las cuenta como a, como b, etc.). Si no hubiera coincidencia se ejecuta el caso default.

Cada case puede tener una o más acciones. La estructura switch es diferente de las otras estructuras en que no se necesitan llaves alrededor de varias acciones de un case de un switch.

4. La estructura de repetición do/while

Es similar a la estructura while. La diferencia es que la condición de continuación del ciclo se testea despues de ejecutar el cuerpo del ciclo y por lo tanto el cuerpo del ciclo se ejecutará al menos una vez.

La forma del enunciado es:

```
do
   enunciado
while (condicion);
```

Cuando termina do/while la ejecución continuará con el enunciado que aparezca despues de la palabra clave while. En el caso de que enunciado sea más de una sentencia se deben colocar entre llaves.

El programa a continuación utiliza una estructura do/while para imprimir los números del 1 al 10.

Programa

```
#include <stdio.h>

main ()
{
      int contador=1;

      do {
         printf (" %d ",contador);
         }
```

```
        while (++contador <= 10);
        system("PAUSE");
}
```

El enunciado:

```
do
   enunciado
while (condicion)
```

es equivalente a :

```
enunciado;
while (condicion)
        enunciado;
```

5. Los enunciados break y continue

5.1. break

Se utilizan para modificar el flujo de control. El enunciado **break**, cuando se utiliza en una estructura while, for, do/while o switch causa la salida inmediata de dicha estructura. La ejecución del programa continua con el primer enunciado después de la estructura.

El programa a continuación muestra el uso del enunciado break es una estructura for.

```
main ()
{
        int x;

        for (x=1; x <= 10; x++)
        {
            if (x == 5) break;
            printf (" %d ",x);
        }
        printf ("\n Salimos del loop con x = %d", x);
        system("PAUSE");
}
```

El programa imprime el 1,2,3,4 y el mensaje Salimos del loop con x = 5.

Programas equivalentes con while y do/while son los siguientes:

```
/* programa con while */
main ()
{
        int x;
```

```
        x=1;
        while (x <= 10)
        {
          if (x == 5) break;
          printf (" %d ",x);
          x++;
        }
        printf ("\n Salimos del loop con x = %d", x);
        system("PAUSE");
}


/* programa con do/while */
main ()
{
        int x;

        x=1;
        do
        {
         if (x == 5) break;
         printf (" %d ",x);
         x++;
        }
        while (x <= 10)
        printf ("\n Salimos del loop con x = %d", x);
        system("PAUSE");
}
```

5.2. continue

El enunciado **continue** cuando se ejecuta en una estructura while, for, do/while, salta los enunciados restantes del cuerpo de dicha estructura y ejecuta la siguiente iteración del ciclo.

En las estructuras while y do/while la prueba de continuación del ciclo se evalúa de inmediato luego del continue. En la estructura for se ejecuta la expresión incremental y a continuación se evalúa la prueba de continuación del ciclo.

A continuación veremos un programa que utiliza continue en una estructura for.

```
main ()
{
        int x;

        for (x=1; x <= 10; x++)
        {
         if (x == 5) continue;
```

```
        printf (" %d ",x);
       }
       printf ("\n Salteamos el valor 5 con continue");
       system("PAUSE");
}
```

este programa imprime:

```
1 2 3 4 6 7 8 9 10
Salteamos el valor 5 con continue
```

programas equivalentes con while y do/while son:

```
/* programa con while */
main ()
{
       int x;

       x=1;
       while (x <= 10)
       {
        if (x == 5) { x++; continue;}
        printf (" %d ",x);
        x++;
       }
       printf ("\n Salteamos el valor 5 con continue");
       system("PAUSE");
}
```

```
/* programa con do/while */
main ()
{
       int x;

       x=1;
       do
       {
        if (x == 5) { x++; continue;}
        printf (" %d ",x);
        x++;
       }
       while (x <= 10);

       printf ("\n Salteamos el valor 5 con continue");
       system("PAUSE");
}
```

6. Ejemplos de programas simples con for y do/while

6.1. Factorial

Programa

```
/* Factorial con do/while */
main ()
{
      int i,n,fac;
      printf("Ingrese el numero del cual calculara factorial \n");
      scanf(" %d", &n);
      i=1;
      fac=1;
      do
          {
            fac*=i;
            i++;
          }
      while (i <= n);
      printf ("Factorial de %d es igual a %d",n,fac);
}


/* Factorial con for */
main ()
{
      int n,fac=1;
      printf("Ingrese el numero del cual calculara factorial \n");
      scanf(" %d", &n);
      for (int i=1; i <= n; fac*=i;i++);
      printf ("Factorial de %d es igual a %d \n",n,fac);
      system("PAUSE");
}
```

6.2. Potencia n-esima

Programa

```
/* Calculo la potencia n-esima de un numero dado con do/while*/
main ()
{
      int i,n,num,potencia;
/* num guarda el numero cuya potencia se calculara,
n guarda la potencia,
potencia se utiliza para calcular la potencia,
i cuenta la cantidad de veces que he multiplicado potencia por num */

      i=1;
```

```
        potencia=1;
        printf (“ Ingrese potencia \n”);
        scanf(“%d”, &n);
        printf (“ Ingrese numero cuya potencia desea calcular \n”);
        scanf (“%d,&num);
        do
        {
          potencia *= num;
          i++;
        }
        while (i¡=n);
        printf (“Potencia %d de %d es igual a %d \n”,n,num,potencia);
}
```

Programa

```
/* Calculo la potencia n-esima de un numero dado con for */
main ()
{
        int n,num,potencia;
/* num guarda el numero cuya potencia se calculara,
n guarda la potencia,
potencia se utiliza para calcular la potencia,
i cuenta la cantidad de veces que he multiplicado potencia por num */

        potencia=1;
        printf (“ Ingrese potencia \n”);
        scanf(“%d”, &n);
        printf (“ Ingrese numero cuya potencia desea calcular \n”);
        scanf (“%d,&num)
;       for (int i=1; i <= n; potencia*=num,i++) ;
        printf (“Potencia %d de %d es igual a %d \n”,n,num,potencia);
}
```

7. Casos de estudio de programación con do/while y for

7.1. Repetición controlada por contador

Promedio de la nota de un examen de una clase con 10 alumnos

Programa

```
/* Promedio programado con do/while */
main ()
{
        total=0;
```

```
        nro_alumno=1;
        do
        {
         scanf(" %d",&calificacion);
         total+=calificacion;
         nro_alumno++;
        }
        while (nro_alumno <= 10);
        promedio=total/10;
        printf("El promedio es %f",promedio);
        system("PAUSE");
}
```

Programa

```
/* Promedio programado con for */
main ()
{
        total=0;
        nro_alumno=1;
        for (nro_alumno=1;nro_alumno <= 10; nro_alumno++)
            {
             scanf(" %d",&calificacion);
             total+=calificacion;
            }
        promedio=total/10;
        printf("El promedio es %f",promedio);
        system("PAUSE");
}
```

7.2. Repetición controlada por centinela

El problema es igual al anterior salvo que en vez de considerar 10 estudiantes consideramos una cantidad arbitraria.

Ingresaremos datos y cuando hayamos terminado con todos los datos ingresaremos -1.

Hay casos por ejemplo éste en el cual el problema se puede resolver de una forma directa con while y no hay una forma directa de resolverlo con do/while. En este caso resolvemos el problema utilizando un if para diferenciar cuando no hay datos de cuando los hay.

Programa

```
/* Programa con do/while */
main ()
{
        int total=0;
        int cant_alumnos=0;
        int centinela;
        int calificacion;
```

```
printf("Ingrese -1 para terminar, cualquier otro valor para continuar\n");
scanf(" %d",&centinela);
if (centinela!=-1)
  do
  {
   scanf(" %d", &calificacion);
   total+=calificacion;
   cant_alumnos++;
   printf("Ingrese -1 para terminar, cualquier otro valor para continuar\n");
   scanf(" %d", &centinela);
  }
  while (centinela != -1);
if (cant_alumnos!=0)
{
  promedio=total/cant_alumnos;
  printf("El promedio es %f", promedio);
}
else
  printf("No se ingresaron calificaciones");
system("PAUSE");
```

Programa

```
/* Programa con for */
main ()
{
    int total=0,centinela,calificacion;
    printf("Ingrese -1 para terminar, cualquier otro valor para continuar\n");
    scanf(" %d",&centinela);
    for(int cant_alumnos=0;centinela!=-1;cant_alumnos++)
      {
       scanf(" %d", &calificacion);
       total+=calificacion;
       printf("Ingrese -1 para terminar, cualquier otro valor para continuar\n");
       scanf(" %d",&centinela);
      }
    if (cant_alumnos!=0)
    {
      promedio=total/cant_alumnos;
      printf("El promedio es %f", promedio);
    }
    else
      printf("No se ingresaron calificaciones");
    system("PAUSE");
```

7.3. Estructuras de control anidadas

Consideremos el problema:

Una universidad ofrece un curso que prepara alumnos para un examen. La universidad desea saber que tan bien salieron sus alumnos en el examen. Se ingresará un 1 si el alumno pasó el examen y un 2 si lo reprobó. Se quiere saber total de aprobados y total de reprobados en un total de 10 alumnos.

Programa

```
/* Programa con do/while*/

main ()
{
      int aprobados=0;
      int reprobados=0;
      cant_alumnos=1;
      do
          {
           scanf(" %d", &calificacion);
           if (calificacion==1) aprobados++;
           else reprobados++;
           cant_alumnos++;
          }
      while (cant_alumnos <= 10);
      printf("Total de aprobados");
      printf(" %d\n", aprobados);
      printf("Total de reprobados");
      printf(" %d", reprobados);
      system("PAUSE");
```

Programa

```
/* Programa con for */

main ()
{
      int aprobados=0;
      int reprobados=0;
      for (int cant_alumnos=1; cant_alumnos <= 10; cant_alumnos ++)
          {
           scanf(" %d", &calificacion);
           if (calificacion==1) aprobados++;
           else reprobados++;
          }
      printf("Total de aprobados");
      printf(" %d\n", aprobados);
      printf("Total de reprobados");
      printf(" %d", reprobados);
      system("PAUSE");
```

8. Una colección de programas utiles

8.1. Copia de entrada en la salida con do/while y con for

Programa

```
main()
{
        int c;

        do
        {
            c=getchar();
            if (c==EOF) break;
            else putchar(c);
        }
        while (c!=EOF);
}
```

Hemos escrito el programa arriba usando break, otra forma es imprimiendo el caracter leido si es distinto de eof como a continuación:

Programa

```
main()
{
        int c;

        do
        {
            c=getchar();
            if (c!=EOF) putchar(c);
        }
        while (c!=EOF);
}
```

Podemos tambien escribir el programa asociando la condición del do/while con el getchar() como a continuación:

Programa

```
main()
{
        int c;

        c=getchar();
        do
            if (c!=EOF) putchar(c);
        while((c=getchar())!=EOF);
```

```
}
```

otra forma (que queda de ejercicio) es colocar la instrucción if antes del do/while de forma que se ejecute el do/while si c!=EOF.

A continuación presentamos el programa con for:

Programa

```
main()
{
        int c;

        for (;c=getchar()!=EOF;putchar(c));
}
```

8.2. Cuenta de la cantidad de caracteres leidos

Programa

```
/* Programa con do/while */
main()
{
      long nc;

      nc=0;
      if (getchar()!=EOF)
        do
            nc++;
        while (getchar()!=EOF);
      printf("Cantidad de caracteres leidos = %ld\n", nc);
      system("PAUSE)";
}
```

Programa

```
/* Programa con do/while */
main()
{
      long nc;

      nc=0;
      for(;getchar()!=EOF;nc++) ;
      printf("Cantidad de caracteres leidos = %ld\n", nc);
      system("PAUSE");
}
```

8.3. Cuenta de la cantidad de lineas leidas

Programa

```
/* Programa con do/while */
main ()
{
      int c,nl;

      nl=0;
      if ((c=getchar())!=EOF)
        do
            if (c=='\n') nl++;
        while ((c=getchar())!=EOF);
      printf("Cantidad de nueva linea %d\n", nl);
      system("PAUSE");
}
```

Programa

```
/* Programa con for */
main ()
{
      int c,nl;

      for (nl=0; (c=getchar())!=EOF;)
          if (c=='\n') nl++;
      printf("Cantidad de nueva linea %d\n", nl);
      system("PAUSE");
}
```

8.4. Cantidad de nueva linea, palabras y caracteres con do/while y for

Cada vez que el programa encuentra el primer caracter de una palabra lo cuenta. La variable en_palabra recuerda cuando el programa está dentro de una palabra y cuando no. Inicialmente no lo es. Cuando no está dentro de una palabra y el siguiente caracter no es espacio, nueva linea ni tabulador seteamos que está dentro de una palabra. Se suma uno a cantidad de palabras.

Programa

```
main()
{
      int c, nl, nw, nc, en_palabra;

      en_palabra = false;
      nl=nw=nc=0;
      if ((c=getchar())!=EOF)
```

```
        do
        {
          ++nc;
          if (c=='\n') ++nl;
          if (c==' ' || c=='\n' || c=='\t') en_palabra=false;
          else if (!en_palabra) { en_palabra=true; ++nw; }
        }
        while ((c=getchar())!=EOF);
      printf(" %d %d %d \n", nl, nw, nc);
      system("PAUSE");
}
```

Principios de Programación

El lenguaje C

1. Instrucciones de entrada/salida

Una parte importante de la solución de cualquier problema es la presentación de los resultados. Analizaremos a fondo las caracteristicas de formato de printf y scanf.

Ya hemos visto algunas caracteristicas de scanf y printf. Resumiremos esas características y presentaremos otras.

1.1. Flujos

Toda entrada y salida se ejecuta mediante *flujos* (secuencias de caracteres organizadas en lineas). Cada linea está formada de cero o más caracteres y está terminada por el caracter de nueva linea.

Cuando empieza la ejecución del programa de forma automática se conectan tres flujos al programa. Por lo regular el *flujo estandar de entrada* se conecta al teclado y el *flujo estandar de salida* se conecta a la pantalla. Un tercer flujo, el *error estandar* se conecta a la pantalla. Los mensajes de error son extraidos o desplegados al flujo estandar de error.

Con frecuencia los sistemas operativos permiten que estos flujos sean redirigidos a otros dispositivos.

stdin y stdout pueden redirigirse a archivos o conductos (pipes), por ejemplo:

prog < infile

hace que prog lea el contenido de infile en vez de leer del teclado. Otro ejemplo:

prog | otroprog

hace que la salida estandar de prog sea la entrada estandar de otroprog.

1.2. fgetc, fputc, getchar, putchar, fscanf, fprintf

Cuando un programa comienza, hay 3 archivos que se abren automáticamente y proveen punteros a archivos: los apuntadores de archivo asignados a la entrada estandar (**stdin**) , a la salida estandar (**stdout**) y al error estandar (**stderr**). La idea es que cuando quiero leer o imprimir en las entrada y salida estandar coloco como nombre de archivo stdin y stdout.

Puedo sino leer o imprimir a archivos (en cuyo caso doy el nombre del archivo en la intrucción de entrada/salida).

La biblioteca estandar provee funciones para leer datos de archivos y para escribir datos a los archivos.

Los flujos proporcionan canales de comunicación entre archivos y programas. Abrir un archivo regresa un apuntador a una estructura FILE (FILE es

un nombre de tipo definido en stdio) que contiene información utilizada para procesar dicho archivo.

La operación **fgetc**, declarada como:

int fgetc(FILE *stream)

devuelve el siguiente caracter del flujo de entrada al cual señala stream. Si el flujo está al fin de archivo fgetc devuelve EOF. Idem si hay un error de lectura (devuelve EOF).

Podemos definir getchar() en terminos de getc con la sentencia:

#define getchar() fgetc(stdin)

La operación **fputc**, declarada como:

int fputc(int c,FILE *stream)

escribe el caracter c en el flujo de salida stream. Si hay un error de escritura devuelve EOF.

Podemos definir putchar(c) en terminos de fputc con la sentencia:

#define putchar(c) fputc(c,stdout)

Trabajaremos con la entrada y salida estandar.

2. Salida con formato utilizando printf

Cada llamada a printf contiene una *cadena de control de formato* que describe el formato de la salida. La cadena de control de formato consiste de especificadores de conversión, banderas, anchos de campo, precisiones y caracteres literales.

La función printf puede llevar a cabo las siguientes capacidades de formato:

1. Redondear valores de punto flotante a un número indicado de valores decimales.
2. Alinear una columna de números con puntos decimales apareciendo uno por encima del otro.
3. Salidas justificadas a la derecha o a la izquierda.
4. Insertar caracteres literales en posiciones precisas en una linea de salida.
5. Representación en formato exponencial de números de punto flotante.
6. Representación en formato octal y hexadecimal de enteros no signados.
7. Despliegue de todo tipo de datos con anchos de campo de tamaño fijo y precisiones.

La forma de la operación **printf** es la siguiente:

printf(cadena de control de formato, otros argumentos)

los otros argumentos son opcionales y se corresponden con las especificaciones de conversión, las que comienzan con % y forman parte de la cadena de control de formato. La cadena de control de formato se coloca entre comillas.

2.1. Como imprimir enteros

A continuación se describen cada uno de los especificadores de conversión de enteros.

d	Despliega un entero decimal signado
i	Muestra un entero decimal signado. i y d son diferentes cuando se usan con scanf y son iguales cuando se usan en printf.
o	Muestra un entero octal no signado
u	Muestra un entero decimal no signado
x o X	Muestra un entero hexadecimal no signado. con X se imprimen las letras A hasta F en mayuscula, con x se imprimen las mismas letras en minuscula.
h o l	se coloca antes de cualquier especificador de conversión de enteros para indicar que se muestra un entero short o long respectivamente.

Abajo presentamos un programa ejemplo que muestra el uso de los especificadores de conversión. Notar que solo se imprime el signo menos (-), más adelante veremos como imprimir los signos +. Notar también que el uso de u en un número negativo cambia su valor.

Programa

```
#include <stdio.h>

main ()
{
      printf(“ %d\n”,455);
      printf(“ %i\n”,455);
      printf(“ %d\n”,+455);
      printf(“ %d\n”,-455);
      printf(“ %hd\n”,32000);
      printf(“ %ld\n”,2000000000);
      printf(“ %o\n”,455);
      printf(“ %u\n”,455);
      printf(“ %d\n”,-455);
      printf(“ %x\n”,455);
      printf(“ %X\n”,455);
      system(”PAUSE”); }
```

El programa anterior imprime:

455
455
455
-455
32000
2000000000
707
455
65081 (u otro valor que depende del compilador)
1c7
1C7

Notar que 455 decimal es 707 octal y que 455 decimal es 1c7 hexadecimal.

El uso de hd para valores muy grandes puede dar error (desplega cualquier valor).

2.2. Como imprimir números de punto flotante

Un valor de punto flotante contiene un punto decimal, como en 33.5 o 657.983. Los valores de punto flotante se pueden imprimir en uno de varios formatos.

Los especificadores de conversión son los siguientes:

e o E	Muestra un valor en punto flotante en notación exponencial.
f	Muestra valores en punto flotante.
g o G	Despliega un valor en punto flotante ya sea en la forma de punto flotante f o en la notación exponencial e (o E).
L	Se coloca antes de cualquier especificador de conversión de punto flotante para indicar que está mostrando un valor de punto flotante long double.

En forma preestablecida los valores impresos con los especificadores de conversión e, E y f son extraidos con 6 digitos de precisión a la derecha del punto decimal.

El especificador de conversión f siempre imprime por lo menos un digito a la izquierda del punto decimal (puede ser 0).

Los especificadores de conversión e y E imprimen respectivamente la e en minusculas y la E en mayusculas antes del exponente y siempre se imprime exactamente un digito a la izquierda del punto decimal (distinto de 0).

El especificador de conversión g (G) imprime en formato e (E) o en formato f sin ceros después del punto decimal. Los valores se imprimen con e (E) si despues si despues de convertir el valor a notación exponencial, el exponente del valor es menor a -4 o es mayor o igual a la precisión especificada (por omisión en el caso de g o G, 6 dígitos significativos). De lo contrario se utilizará f.

Si se utiliza g o G los valores 0.0000875, 8750000.0,8.75,87.50 y 875 se imprimen como: 8.75e-005,8.75e+006,8.75,87.5 y 875.

La precisión de los especificadores de conversión g y G indican el número máximo de digitos significativos impresos incluyendo el dígito a la izquierda del punto decimal. Cuando decimos la precisión es de 6 dígitos significativos, nos referimos a 5 dígitos despues del punto decimal y 1 a la izquierda del punto decimal. Por ejemplo, si quisiera imprimir 1234567.89 con %g se imprime: 1.23457e+006 (se corre el punto decimal 6 posiciones y se imprimen 5 decimales), si quisiera imprimir 12345678.9 con %g se imprime: 1.23457e+007 (se corre el punto decimal 7 posiciones y se imprimen 5 decimales). Si quisiera imprimir 1234.567 se imprime 1234.57.

Considere el siguiente programa:

```
#include <stdio.h>

main ()
{
      printf(“ %e\n”, +1234567.89);
      printf(“ %E\n”, -1234567.89);
      printf(“ %e\n”, 12.3456789e+05);
      printf(“ %f\n”, 1234567.89);
      printf(“ %f\n”, 12.3456789e+05);
      printf(“ %g\n”, 1234567.89);
      printf(“ %G\n”, 12.3456789);
}
```

se imprime:

```
1.234568e+006
-1.234568E+006
1.234568+006
1234567.890000
1234567.890000
1.23457e++006
12.3457
```

2.3. Cómo imprimir caracteres

Para imprimir caracteres individuales se utiliza el especificador de conversión ” %c”. Se requiere de un argumento char.

Ejemplo

```
main ()
{
      char caracter=’A’;
      printf(“ %c\n”,caracter);
      system(“PAUSE”);
}
```

imprime el caracter 'A'.

2.4. Cómo imprimir con anchos del campo y precisiones

El tamaño exacto de un campo en el cual se imprimen datos se especifica por el ancho del campo. Si el ancho del campo es mayor que los datos que se están imprimiendo, a menudo los datos dentro de dicho campo quedarán justificados a la derecha. Un entero que representa el ancho del campo es insertado en la especificación de conversión, entre el signo de % y el especificador de conversión.

En el caso de que el ancho del campo es menor que los datos que se están imprimiendo este es aumentado automáticamente. Además el signo de menos de un valor negativo ocupa una posición.

Los anchos de campo pueden ser utilizados con todos los especificadores de conversión.

La función printf también da la capacidad de definir la **precisión** con la cual los datos se imprimirán. La precisión tiene significados distintos para diferentes tipos de datos. Cuando se utiliza con especificadores de conversión de enteros, la precisión indica el número mínimo de digitos a imprimirse. Si el valor impreso contiene menos dígitos que la precisión especificada al valor impreso se le antepondrán ceros hasta que el número de digitos sea igual a la precisión. La precisión por omisión para enteros es 1.

Cuando se utiliza con especificadores de conversión de punto flotante e,E y f la precisión es el número de dígitos que aparecerá despues del punto decimal. Cuando se utilice con los especificadores de conversión g y G, la precisión es el número máximo de digitos significativos a imprimirse (se consideran digitos a la izquierda del punto decimal más digitos a la derecha).

Para especificar la precisión, coloque un punto decimal seguido por un entero que representa la precisión entre el ancho del campo y el especificador de conversión. Cuando un valor en punto flotante se imprime con una precisión menor que el número original de decimales el valor se redondea.

Ejemplo:

printf(“%9.3f”,123.456789);

Imprime 123.457 justificado a la derecha en un campo de 9 dígitos.

Utilizando expresiones enteras en la lista de argumentos a continuación de la cadena de control de formato es posible especificar el ancho de campo y la precisión. Para utilizar esta característica inserte un asterisco (*) en el lugar del ancho de campo y/o la precisión. El argumento coincidente en la lista de argumentos se evalua y se utiliza en lugar del asterisco. El valor correspondiente al ancho de campo puede ser negativo, el correspondiente a la precisión debe ser positivo. Un valor negativo para el ancho de campo hace que la salida se justifique a la izquierda.

Ejemplo:

printf(“%*.*f”,7,2,98.736);

utiliza 7 para el ancho de campo y 2 para la precisión. Imprime 98.74 justificado a la derecha.

Ejemplo:

```
#include <stdio.h>


main ()
{
/* Uso de ancho del campo */
      printf(“%4d\n”, 12);
      printf(“%4d\n”, -123);
      printf(“%4d\n”, 123456);
      printf(“%3f\n”, 12.34);
      printf(“%15f\n”, 12.34);
/* Uso de precision */
      printf(“%.4d\n”, 873);
      printf(“%.9d\n”, 873);
      printf(“%5.4d\n”, 12);
      printf(“%8.5d\n”, 123);
      printf(“%4.5d\n”, 123);
      printf(“%9.3f\n”, 1234.5678);
      printf(“%20.3e\n”, 1234.5678);
      printf(“%5.4g\n”, 1234.5678);
      printf(“%4.4g\n”, 111.6);
```

imprime: (b significa espacio)

```
bb12
-123
123456
12.340000
bbbbbb12.340000
0873
000000873
b0012
bbb00123
00123
b1234.568
bbbbbbbbbb1.235e+003
b1235
111.6
```

2.5. Uso de banderas en la cadena de control de formato de printf

La función printf tiene también *banderas* para complementar sus capacidades de formato de salida. Están disponibles cinco banderas, para uso en cadenas de control de formato. Las banderas son las siguientes:

-	signo de menos. Justificación de la salida a la izquierda. El signo de menos se coloca entre el
+	signo de más. Despliega un signo de más antes de valores positivos y un signo de menos antes de valores negativos.
espacio	Imprime un espacio antes de un valor positivo que no se imprima con la bandera +.
#	Antecede un 0 al valor extraido cuando se utiliza con el especificador de conversión octal o. Antecede 0x o 0X al valor de salida cuando se utiliza con los especificadores de conversión hexadecimales x o X. Obliga a un punto decimal para un número de punto fl otante impreso con e,E,f,g o G que no contenga una parte fraccionaria (se imprimen 0's a la derecha del punto decimal). En el caso de los especificadores g y G no se eliminan los ceros a la derecha.
0	cero. Rellena un campo con ceros a la izquierda.

Las banderas se colocan de inmediato a la derecha del signo de %. En una especificación de conversión se pueden conmbinar varias banderas.

Ejemplo 1

```
main ()
{
      printf(" %5d %15f\n", 22, 12.34);
      printf(" % -5d % -15f\n",22,12.34);
      system("PAUSE");
}
```

imprime:

```
bbb22bbbbbbb12.340000
b22bbb12.340000
```

observar en el ejemplo anterior además del uso del menos (-) el uso del espacio.

Ejemplo 2

```
main ()
{
	printf(" %d\n %d\n", 786,-786);
	printf(" %+d\n %+d\n", 786, -786);
	system("PAUSE");
}
```

se imprime:

```
786
-786
+786
-786
```

Ejemplo 3

```
main ()
{
	int c= 1427;
	float p= 235.7;
	float q= 123;

	printf(" %#o\n", c);
	printf(" %#x\n", c);
	printf(" %#X\n", c);
	printf("\n %g\n", p);
	printf(" %#g\n", p);
	printf(" %#g\n", q);
	system("PAUSE");
}
```

se imprime:

```
02623
0x593
0X593

235.7
235.700
123.000
```

recordar que g o G imprime sin ceros despues del punto decimal. Si queremos que no se eliminen los ceros a la derecha debemos usar #. Observar que g se

declaro sin punto decimal ni ceros a la derecha y cuando se imprime se imprime con punto decimal y ceros. Pasa lo mismo si imprimiera con f o con e (en este ultimo caso se usa la notación exponencial).

Ejemplo 4

```
main ()
{
      printf(" %+09d\n", 452);
      printf(" %09d\n", 452);
      system("PAUSE");
}
```

se imprime:

```
+00000452
000000452
```

3. Entrada con formato utilizando scanf

Cada enunciado scanf contiene una cadena de control de formato que describe el formato de los datos que se leen. La cadena de control está formada de especificaciones de conversión y de caracteres literales. La función scanf tiene las siguientes capacidades de formato de entrada:

1. Entrada de todo tipo de datos.
2. Entrada de caracteres especificos desde un flujo de entrada.
3. Omitir caracteres especificos del flujo de entrada.

La función scanf se escribe en la forma siguiente:

```
scanf(string-de-control-de-formato, otros-argumentos)
```

el primer argumento describe los formatos de la entrada y los otros-argumentos son apuntadores a variables en las cuales se almacena la entrada.

A continuación se resumen los especificadores de conversión utilizados para introducir todos los tipos de datos.

Enteros

d	Lee un entero decimal, opcionalmente signado. El argumento correspondiente es un apuntador a un entero.
i	Lee un entero decimal, octal o hexadecimal, opcionalmente signado. El argumento correspondiente es un apuntador a un entero.
o	Lee un entero octal. El argumento correspondiente es un apuntador a un entero no signado.
u	Lee un entero decimal no signado. El argumento correspondiente es un apuntador a un entero no signado.
x o X	Lee un entero hexadecimal. El argumento correspondiente es un apuntador a un entero no signado.
h o l	Se coloca antes de cualquiera de los especificadores de conversión de enteros, para especificar que un entero short o long será introducido.

Números de punto flotante

e, E, f, g o G	Lee un valor en punto flotante. El argumento correspondiente es un apuntador a una variable de punto flotante.
L	Se coloca delante de cualquier especificador de conversión de punto flotante para indicar que un valor double o long double será introducido.

Caracteres

c	Lee un caracter. El argumento correspondiente es un apuntador a char.

EJEMPLOS

Ejemplo 1

```
main ()
{
      int a,b,c,d,e,f,g;

      printf("Entre siete enteros:\n");
      scanf(" %d %i %i %i %o %u %x", &a, &b, &c, &d, &e, &f, &g);
      printf("La entrada desplegada como enteros decimales es:\n");
      printf(" %d %d %d %d %d %d %d\n", a, b, c, d, e, f, g);
      system("PAUSE");
}
```

lee e imprime lo siguiente:

```
Entre siete enteros:
-70 -70 070 0x70 70 70 70
La entrada desplegada como enteros decimales es:
-70 -70 56 112 56 70 112
```

El primer $\%d$ lee -70.
El primer $\%i$ lee -70.
El segundo $\%i$ lee 070 que es 70 octal que es 56 decimal.
El tercer $\%i$ lee 0x70 que es 70 hexadecimal que es 112 decimal.
El $\%o$ lee 70 octal que es 56 decimal.
El $\%u$ lee 70 sin signo que es 70 decimal.
El $\%x$ lee 70 hexadecimal que es 112 decimal.

Ejemplo 2

```
main ()
{
      float a,b,c;

      printf("Entre tres numeros punto flotante:\n");
      scanf(" %e %f %g", &a, &b, &c);
      printf("Los numeros en notacion punto flotante son:\n");
      printf("%f\n %f\n %f\n", a, b, c);
      system("PAUSE");
}
```

lee e imprime lo siguiente:

```
Entre tres numeros punto flotante:
1.27987 1.27987e+003 3.38476e-006
Los numeros en notacion punto flotante son:
1.279870
1279.869995
0.000003
```

En la especificación de conversión scanf se puede utilizar un ancho de campo para leer un número especifico de caracteres a partir del flujo de entrada. Por ejemplo:

```
main ()
{
      int x, y;
      printf("Entre un entero de 6 digitos\n");
      scanf("%2d %d", &x, &y);
      printf("Los enteros ingresados son %d y %d\n",x,y);
      system("PAUSE");
}
```

La entrada/salida de este programa es:

```
Entre un entero de 6 digitos
123456
Los enteros ingresados son 12 y 3456
```

3.1. Cómo saltear caracteres de la entrada

A menudo es necesario omitir ciertos caracteres del flujo de entrada. Por ejemplo una fecha podría ser introducida como

7-9-91

Queremos guardar el dia, mes, año y saltear los guiones de la entrada. Para eliminar caracteres innecesarios los incluimos en la cadena de control de formato de scanf junto con *. Por ejemplo, leemos la fecha con:

scanf(" %d %*c %d %*c %d", &dia, &mes, &anio);

El caracter de supresión de asignación le permite a scanf leer cualquier tipo de datos a partir de la entrada y descartarlos sin asignarlos a una variable. Por ejemplo la instrucción:

scanf(" %2d %f %*d", &i, &x);

con la entrada:

56789 0123 a

asignará 56 a i, 789.0 a x y saltea 0123. La proxima llamada a una instrucción de entrada leerá la letra 'a'.

3.2. Ancho de campo en la lectura de reales

Podemos asociar un ancho de campo a los valores reales leidos. El ancho abarca la parte entera, el punto y los decimales. Por ejemplo:

```
scanf(" %4f%2f, &f1, &f2);
printf(" %f %f", f1, f2);
```

cuando introduzco la entrada:

12.34 12.0

imprimirá

12.300000 4.000000

idem si utilizo e, E, g o G.

Principios de Programación

El lenguaje C

1. Introducción, Funciones

La mayor parte de los programas de computo que resuelven problemas de la vida real son mucho mayores que los programas que hemos visto.

La experiencia ha mostrado que la mejor forma de desarrollar y mantener un programa grande es construirlo a partir de piezas menores o módulos, siendo cada uno de ellos más facil de manipular que el programa original. Esta técnica se conoce como *divide y venceras* .

C fue diseñado para hacer funciones eficientes y faciles de usar. Los programas C consisten generalmente de varias funciones pequeñas en vez de pocas grandes.

1.1. Módulos de programa en C

Una función es un fragmento de código que realiza una tarea bien definida. Por ejemplo, la función printf imprime por la salida estandar los argumentos que le pasamos.

En C los módulos se llaman funciones. Por lo general en C los programas se escriben combinando nuevas funciones que el programador escribe con funciones "preempacadas"disponibles en la *biblioteca estandar de C*.

La biblioteca estandar de C contiene una amplia colección de funciones para llevar a cabo cálculos matemáticos, manipulaciones de cadenas, entrada/salida, y muchas otras operaciones útiles. Esto facilita la tarea del programador porque estas funciones proporcionan muchas de las capacidades que los programadores requieren.

Las funciones se invocan mediante una *llamada de función*. La llamada de función especifica el nombre de la misma y proporciona información (en forma de *argumentos*) que la función llamada necesita a fin de llevar acabo su tarea.

1.2. Funciones

Las funciones permiten a un programador modularizar un programa.

Todas las variables declaradas en las definiciones de función son *variables locales* (son conocidas sólo 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 funciones. Los parámetros de función son también variables locales.

Existen varios intereses que dan motivo a la "funcionalización"de un programa. El enfoque de divide y venceras hace que el desarrollo del programa sea más manipulable. Otra motivación es la reutilización del software - el uso de funciones existentes como bloques constructivos para crear nuevos programas.

Cada función deberá limitarse a ejecutar una tarea sencilla y bien definida y el nombre de la función deberá expresar claramente dicha tarea.

Si no se puede elegir un nombre consiso es probable que la función esté intentando ejecutar demasiadas tareas diversas.

1.3. Definiciones de función

Cada programa que hemos presentado ha consistido de una función llamada **main** que para llevar a cabo sus tareas ha llamado funciones estandar de biblioteca. Veremos ahora como los programadores escriben sus propias funciones personalizadas.

Considere un programa que utiliza una función **cuadrado** para calcular los cuadrados de los enteros del 1 al 10.

```
/* Funcion que calcula cuadrado de un numero. */

int cuadrado(int);

main ()
{
      int x;

      for (x=1; x <= 10; x++)
          printf(" %d ", cuadrado(x));
      printf ("\n");
      system("PAUSE");
}


/* Definicion de funcion */

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

Se imprime:

1 4 9 16 25 36 49 64 81 100

Es conveniente dejar lineas en blanco entre definiciones de función para mejorar la legibilidad del programa.

La función cuadrado es invocada dentro de printf. Ésta recibe una copia del valor de x en el parámetro y. A continuación calcula y*y. El resultado se regresa a la función printf en main donde se llamo a cuadrado y printf despliega el resultado. Éste proceso se repite diez veces utilizando la estructura de repetición for.

La definición de cuadrado muestra que esta función espera un parámetro entero. La palabra reservada **int** que precede al nombre de la función indica que la función devuelve un resultado entero. El enunciado **return** en la función cuadrado pasa el resultado del cálculo de regreso a la función llamadora.

La linea

```
int cuadrado(int);
```

es un **prototipo de función**. El int dentro del paréntesis informa al compilador que cuadrado espera recibir del llamador un valor entero. El int a la izquierda del nombre de la función le informa al compilador que la función regresa al llamador un resultado entero. El compilador hace referencia al prototipo de la función para verificar que la llamadas contengan el tipo de regreso correcto, el número correcto de argumentos, el tipo correcto de argumentos y el orden correcto de los argumentos.

El formato de una definición de función es:

```
tipo_de_regreso nombre_de_funcion(lista de parametros)
{

        declaraciones

        enunciados
}
```

El nombre de la función es cualquier identificador válido. El tipo de regreso es el tipo del resultado que se regresa al llamador. Se coloca como tipo **void** cuando la función no regresa un valor.

La lista de parámetros consiste en una lista separada por comas que contiene las declaraciones de los parámetros recibidos por la función al ser llamada. Si una función no recibe ningun valor la lista de parámetros es void. Para cada parámetro debe especificarse un tipo.

En el caso en que la función no reciba parametros se coloca void entre parentesis o simplemente los parentesis sin parametros (esto incluye a main).

Las declaraciones junto con los enunciados dentro de las llaves forman el cuerpo de la función. Bajo ninguna circunstancia puede ser definida una función en el interior de otra función.

Existen diferentes formas de regresar el control al punto desde el cual se invocó una función. Si la función no regresa un resultado el control se devuelve cuando se llega a la llave derecha que termina la función o al ejecutar el enuciado *return*. Si la función regresa un resultado el enunciado *return expresion* devuelve el valor de expresión al llamador y termina la ejecución de la función.

Ejemplo 1

En el siguiente ejemplo se utiliza una función **máximo** definida por el programador para determinar y regresar el mayor de tres enteros. Este valor es regresado main mediante un enunciado return.

```
/* Hallo el mayor de tres enteros */

int maximo(int,int,int);

main ()
```

```
{
      int a,b,c;

      printf("Ingrese tres enteros: ");
      scanf(" %d %d %d", &a, &b, &c);
      printf("El maximo es: %d\n", maximo(a,b,c));
      system("PAUSE");
}
/* Definicion de la funcion maximo */
int maximo(int x,int y,int z)
{
   int max=x;

   if (y>max) max=y;
   if (z>max) max=z;
   return max;
}
```

Ejemplo 2

En el siguiente ejemplo se define una función que calcula el **cubo** de un entero. main lee un valor, e imprime su cubo.

```
/* Hallo el cubo de un entero */

int cubo(int);

main ()
{
      int a;

      printf("Ingrese un entero: ");
      scanf(" %d", &a);
      printf("El cubo de %d es: %d\n", a, cubo(a));
      system("PAUSE");
}
/* Definicion de la funcion cubo */
int cubo(int x)
{
   return x*x*x;
}
```

Argumentos formales y reales

Llamamos argumentos formales a los parámetros de la función y reales a los argumentos que aparecen en la llamada a la función.

1.4. Prototipos de funciones

Una de las características más importante del ANSI C es el prototipo de función. El compilador utiliza los prototipos de función para verificar las llamadas de función.

Es conveniente incluir prototipos de función para aprovechar la capacidad de C de verificación de tipo. Se debe utilizar las directrices de preprocesador de C (#include) para obtener los prototipos de funciones definidos en un archivo aparte.

No es necesario especificar nombres de parámetros en los prototipos de función, pero se pueden especificar para mejorar la comprensión del cdigo fuente. Es necesario colocar punto y coma al final de un prototipo de función.

Una llamada de función que no coincida con el prototipo de la función causará un error de sintaxis.

Otra característica importante de prototipos de función es la **coerción** de argumentos, es decir, obligar a los argumentos al tipo apropiado. Estas conversiones pueden realizarse si se siguen las reglas de promoción de C (por ejemplo tengo un parametro real y la función es llamada con un entero).

La conversión de valores a tipos inferiores por lo general resulta en un valor incorrecto.

Si el prototipo de función de una función no ha sido incluido en un programa el compilador forma su propio prototipo utilizando la primera ocurrencia de la función, ya sea la definición de función o una llamada a dicha función.

En los ejemplos anteriores los prototipos son:

```
int cuadrado(int);

int maximo(int,int,int);

int cubo(int);
```

1.5. Devolución de valores

Una función en C slo puede devolver un valor. Para devolver dicho valor, se utiliza la palabra reservada return cuya sintaxis es la siguiente:

return expresion

Donde expresión puede ser cualquier tipo de dato salvo un array o una función. Adems, el valor de la expresión debe coincidir con el tipo de dato declarado en el prototipo de la función. Por otro lado, existe la posibilidad de devolver múltiples valores mediante la utilización de estructuras.

Dentro de una función pueden existir varios return dado que el programa devolverá el control a la sentencia que ha llamado a la función en cuanto encuentre la primera sentencia return.

Si no existen return, la ejecución de la función continúa hasta la llave del final del cuerpo de la función (}). Hay que tener en cuenta que existen funciones que no devuelven ningún valor. El tipo de dato devuelto por estas funciones puede ser void, considerado como un tipo especial de dato. En estos casos, la

sentencia return se puede escribir como return sin expresión o se puede omitir directamente .

Por ejemplo:

```
void imprime_linea()
{
    printf("esta funcion solo imprime esta linea");
    return;
}
```

es equivalente a :

```
void imprime_linea()
{
    printf("esta funcion solo imprime esta linea");
}
```

1.6. Archivos de cabecera

Cada biblioteca estandar tiene un *archivo de cabecera* correspondiente que contiene los prototipos de función de todas las funciones de dicha biblioteca y las definiciones de varios tipos de datos y de constantes requeridas por dichas funciones.

El programador puede crear archivos de cabecera personalizados. Estos deben terminar en .h. Un archivo de cabecera definido por el programador puede ser incluido utilizando #include.

Cuando un programa utiliza un número elevado de funciones, se suelen separar las declaraciones de función de las definiciones de las mismas. Al igual que con las funciones de biblioteca, las declaraciones pasan a formar parte de un fichero cabecera (extensión .h), mientras que las definiciones se almacenan en un fichero con el mismo nombre que el fichero .h, pero con la extensión .c.

1.7. Acceso a una función

Para que una función realice la tarea para la cual fue creada, debemos acceder o llamar a la misma. Cuando se llama a una función dentro de una expresión, el control del programa se pasa a ésta y sólo regresa a la siguiente expresión de la que ha realizado la llamada cuando encuentra una instrucción return o, en su defecto, la llave de cierre al final de la función.

Generalmente, se suele llamar a las funciones desde la función main, lo que no implica que dentro de una función se pueda acceder a otra función.

Cuando queremos acceder a una función, debemos hacerlo mediante su nombre seguido de la lista de argumentos que utiliza dicha función encerrados entre paréntesis. En caso de que la función a la que se quiere acceder no utilice argumentos, se deben colocar los paréntesis vacos.

Cualquier expresión puede contener una llamada a una función. Esta llamada puede ser parte de una expresión simple, como una asignación, o puede ser uno de los operandos de una expresión ms compleja. Por ejemplo:

```
a=cubo(2);
calculo=b+c/cubo(3);
```

Debemos recordar que los argumentos que utilizamos en la llamada a una función se denominan argumentos reales. Estos argumentos deben coincidir en el número y tipo con los argumentos formales o parámetros de la función. No olvidemos que los argumentos formales son los que se utilizan en la definición y/o declaración de una función.

Los argumentos reales pueden ser variables, constantes o incluso expresiones más complejas. El valor de cada argumento real en la llamada a una función se transfiere a dicha función y se le asigna al argumento formal correspondiente.

Generalmente, cuando una función devuelve un valor, la llamada a la función suele estar dentro de una expresión de asignación, como operando de una expresión compleja o como argumento real de otra función.

Sin embargo, cuando la función no devuelve ningún valor, la llamada a la función suele aparecer sola. Por ejemplo:

```
imprime_linea();
```

2. Cómo llamar funciones. Llamadas por valor y por referencia.

Hasta ahora siempre hemos declarado los parámetros de nuestras funciones del mismo modo. Sin embargo, éste no es el único modo que existe para pasar parámetros.

La forma en que hemos declarado y pasado los parámetros de las funciones hasta ahora es la que normalmente se conoce como "por valor". Esto quiere decir que cuando el control pasa a la función, los valores de los parámetros en la llamada se copian a .ºbjetos"locales de la función, estos .ºbjetos"son de hecho los propios parámetros.

Lo veremos mucho mejor con un ejemplo:

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

int funcion(int n, int m);

main ()
{
      int a, b;
      a = 10;
      b = 20;

      printf("valores de a,b = %d , %d", a, b);
```

```
        printf("valor de funcion(a,b)= %d",funcion(a, b));
        printf("valores de a,b = %d , %d", a, b);
        printf("valor de funcion(10,20) = %d", funcion(10,20));
        system("PAUSE");
}

int funcion(int n, int m)
{
    n = n + 2;
    m = m - 5;
    return n+m;
}
```

Bien, qu es lo que pasa en este ejemplo

Empezamos haciendo a = 10 y b = 20, despus llamamos a la función "funcionçon las objetos a y b como parámetros. Dentro de "funcion.esos parámetros se llaman n y m, y sus valores son modificados. Sin embargo al retornar a main, a y b conservan sus valores originales. Por qu?

La respuesta es que lo que pasamos no son los objetos a y b, sino que copiamos sus valores a los objetos n y m.

Pensemos, por ejemplo, en lo que pasa cuando llamamos a la función con parámetros constantes, es lo que pasa en la segunda llamada a "funcion". Los valores de los parámetros no pueden cambiar al retornar de "funcion", ya que esos valores son constantes.

Si los parámetros por valor no funcionasen así, no sera posible llamar a una función con valores constantes o literales.

Cuando los argumentos se pasan en llamada por valor se efectúa una copia del valor del argumento con el cual se invoca y se asigna a la variable local (parámetro) correspondiente al argumento. Las modificaciones que la función realice a la variable correspondiente al parámetro no afectan a las posibles variables con las que se invocó la función.

Si queremos que los cambios realizados en los parámetros dentro de la función se conserven al retornar de la llamada, deberemos pasarlos por referencia. Esto se hace declarando los parámetros de la función como referencias a objetos. Por ejemplo:

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

int funcion(int *n, int *m);

main ()
{
        int a, b;

        a = 10; b = 20;
        printf("valores de a,b = %d , %d", a, b);
        printf("valor de funcion(a,b)= %d",funcion(&a, &b));
```

```
        printf("valores de a,b = %d , %d", a, b);
        /* printf("valor de funcion(10,20) = %d", funcion(10,20));
        es ilegal pasar constantes como parametros cuando estos
        son referencias (1) */
        system("PAUSE");
}
int funcion(int *n, int *m)
{
    n = n + 2;
    m = m - 5;
    return n+m;
}
```

En este caso, los objetos a y b tendrn valores distintos después de llamar a la función, a saber a valdrá 12 y b valdrá 15 . Cualquier cambio de valor que realicemos en los parmetros dentro de la función, se hará también en los objetos de la llamada.

Esto quiere decir que no podremos llamar a la función con parámetros constantes, como se indica en (1). Un objeto constante no puede tratarse como objeto variable.

Cuando un argumento es pasado en llamada por referencia, las modificaciones que la función realice a la variable correspondiente al parámetro se realizan en las posibles variables con las que se invocó la función.

La llamada por valor debería ser utilizada siempre que la función llamada no necesite modificar el valor de la variable original del llamador.

La llamada por referencia debe ser utilizada solo cuando se necesite modificar la variable original.

Veremos otro modo de pasar parámetros por referencia.

2.1. Más ejemplos:

Veamos un ejemplo. Escribamos una función **intercambio** que reciba dos argumentos enteros y los intercambie. Escribamos la función del siguiente modo:

```
void intercambio(int x, int y)
{
      int temp;

      temp=x;
      x=y;
      y=temp;
}
```

los argumentos están pasados por valor, entoces si llamo a la función del siguiente modo : intercambio(a,b), a y b no son modificados.

Veamos como sería la función y su invocación en el caso de llamada por referencia. La función se declara:

```
void intercambio(int *x, int *y)
```

y en la invocación de la función utilizamos &, en este ejemplo:

```
intercambio(&a,&b);
```

Hay otra forma de realizar llamada por referencia, es colocando en la declaración de la función & del siguiente modo:

```
void intercambio(int &x, int &y)
```

y en la invocación de la función no necesitamos &, en este ejemplo la invocación sería:

```
intercambio(a,b)
```

3. Generación de números aleatorios

Desarrollaremos un programa para ejecución de juegos que incluye varias funciones.

Utilizaremos la función rand() existente en la biblioteca estandar de C para generar números aleatorios. Considere el enunciado:

```
i=rand();
```

Esta función genera un entero entre 0 y RAND_MAX. Éste último valor debe ser por lo menos 32767 que es el valor máximo de un entero de dos bytes.

Si rand en verdad produce enteros aleatorios cualquier número entre 0 y RAND_MAX tiene la misma oportunidad (o probabilidad) de ser elegido cada vez que rand es llamado.

Para generar enteros aleatorios entre 0 y n (donde n es un valor menor que RAND_MAX) tomamos módulo n del entero devuelto por rand().

Comencemos desarrolando un programa para simular 20 tiradas de un dado de 6 caras, e imprimamos el valor de cada tirada. El prototipo de rand se encuentra en stdlib.h.

El programa es el siguiente:

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

main ()
{
      int i;

      for (i=1;i <= 20; i++)
      {
          printf(" %10d", 1 + (rand() % 6));
          if (i % 5 == 0) printf("\n");
      }
}
```

se imprimen 5 números por linea (para ello se utiliza el if de dentro del for).

Para ver que los números ocurren aproximadamente con la misma probabilidad simularemos 6000 tiradas de un dado. Cada entero del 1 al 6 debe ocurrir aproximadamente 1000 veces.

Definiremos un programa que tira el dado 6000 veces y cuenta la cantidad de veces que aparece cada número.

El programa es el siguiente:

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

main ()
{
      int valor, cant, frecuencia1=0; frecuencia2=0,
      frecuencia3=0, frecuencia4=0, frecuencia5=0, frecuencia6=0;

      for (cant=1;cant<=6000;cant++)
      {
          valor=1 + rand() % 6;
          switch (valor)
          {
                  case 1: ++frequencia1; break;
                  case 2: ++frequencia2; break;
                  case 3: ++frequencia3; break;
                  case 4: ++frequencia4; break;
                  case 5: ++frequencia5; break;
                  case 6: ++frequencia6; break;
          }
      }
      printf(" %s %13s\n", "Numero", "Frecuencia");
      printf("1 %13d\n", frecuencia1);
      printf("2 %13d\n", frecuencia2);
      printf("3 %13d\n", frecuencia3);
      printf("4 %13d\n", frecuencia4);
      printf("5 %13d\n", frecuencia5);
      printf("6 %13d\n", frecuencia6);
}
```

El resultado de una ejecución del programa anterior da:

Numero	Frecuencia
1	987
2	984
3	1029
4	974
5	1004
6	1022

Advertir que en el switch anterior no se necesita default.

Si ejecutamos la función que tira 20 dados por segunda vez, aparece impresa exactamente la misma salida. La función rand de hecho genera números pseudoaleatorios. Cada vez que el programa se ejecute el resultado es el mismo.

Existe otra función estandar de biblioteca **srand**. Ésta función toma un argumento entero unsigned para que en cada ejecución del programa, rand produzca una secuencia diferente de números aleatorios.

Veamos el programa anterior de nuevo:

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

main ()
{
      int i;
      unsigned x;

      printf("Ingrese numero: ");
      scanf(" %u", &x);
      srand(x);

      for (i=1;i <= 10; i++)
      {
          printf(" %10d", 1 + (rand() % 6));
          if (i % 5 == 0) printf("\n");
      }
}
```

en el programa, srand toma un valor x como argumento. El prototipo de srand se encuentra en <stdlib.h>. A distintos valores de x corresponden distintos números en rand().

3.1. Ejemplo: un juego de azar

Las reglas del juego son las siguientes:

Un jugador tira dos dados. Cada dado tiene 6 caras. Las caras contienen 1,2,3,4,5 y 6 puntos. Una vez que los dados se hayan detenido se calcula la suma de los puntos en las dos caras superiores. Si a la primera tirada la suma es 7 o bien 11 el jugador gana. Si en la primera tirada la suma es 2, 3 o 12 el jugador pierde (gana la casa). Si en la primera tirada la suma es 4,5,6,8,9 o 10 el jugador debe seguir jugando hasta que o bien se repite dicho número o hasta que salga 7. En el primer caso el jugador gana en el segundo pierde.

El siguiente programa simula este juego. Note que el jugador debe tirar dos dados en cada tirada. Definimos una funcion **tirada** para tirar los dados, calcular e imprimir su suma. Esta función no tiene argumentos pero regresa un entero que es la suma de los dos dados.

El jugador puede ganar o perder en cualquier tirada. Se define una variable **estado** que lleva registro de esto.

Cuando se gana el juego estado se setea en 1. Cuando se pierde el juego, estado se setea en 2. De lo contrario estado se setea a 0 y el juego debe continuar.

El programa es el siguiente:

```
int tirada(void);

main ()
{
      int estado, suma, resultado;
      srand(time(NULL));
      suma = tirada();
      switch(suma)
      {
       case 7 : case 11 : estado=1; break;
       case 2 : case 3 : case 12 : estado=2; break;
       default: estado=0; resultado=suma;
                printf("Tirada dio %d\n",resultado);
                break;
      }
      while (estado==0)
      {
       suma=tirada();
       if (suma == resultado) estado=1;
       else if (suma==7) estado=2;
      }
      if (estado==1) printf("Jugador gana\");
      else printf("Jugador pierde\n");
}

int tirada(void)
{
    int dado1,dado2,suma;

    dado1 = 1 + (rand() % 6);
    dado2 = 1 + (rand() % 6);
    suma = dado1 + dado2;
    printf("Jugador sumo %d + %d = %d\n", dado1, dado2, suma);
    return suma;
}
```

La instrucción srand(time(NULL)) hace que la computadora lea su reloj para obtener automaticamente un valor distinto en el mismo dia. La función time devuelve la hora actual del dia en segundos. Este valor es convertido a un entero unsigned y utilizado en la generación de números aleatorios. La función time toma NULL como argumento (time puede proporcionar al programador

una cadena representando la hora del dia, NULL deshabilita esta capacidad). El prototipo correspondiente a time se encuentra en <time.h>.

Veamos el juego. Despues de la primera tirada, si el jugador gano o perdio (el juego se termino) la estructura while es salteada. El programa continúa con la estructura if/else que imprime si el jugador gano o perdio.

Después de la primera tirada, si el juego no ha terminado, guardamos la suma en resultado. La ejecución continua con la estructura while pues el estado es 0. Dentro del while se tiran los dados, si la suma de estos coincide con resultado el estado se pone en 1, si coincide con 7 el estado se pone en 2, sino continuará la ejecución del while hasta que el jugador gane o pierda. Finalmente se imprimira si el jugador ganó o perdió.

Principios de Programación

El lenguaje C

1. Arreglos

Los **arreglos** son estructuras de datos consistentes en un conjunto de datos del mismo tipo. Los arreglos tienen un tamaño que es la cantidad de objetos del mismo tipo que pueden almacenar. Los arreglos son entidades estáticas debido a que se declaran de un cierto tamaño y conservan éste todo a lo largo de la ejecución del programa en el cual fue declarado.

Decimos arreglo o array indistintamente.

1.1. Declaración

Ejemplo de declaración:

```
int arreglo1[30]
```

declara que arreglo1 es un arreglo que puede contener 30 enteros.

```
#define TAMANIO 100

int arreglo2[TAMANIO]
```

declara que arreglo2 es un arreglo que puede contener TAMANIO enteros. La ventaja de esta última declaración es que todos los programas que manipulen arreglo2 utilizarán como tamaño TAMANIO y si quiero cambiar el tamaño del array alcanza con cambiar la definicion de TAMANIO.

Observar que en la declaración se especifica: tipo de los elementos, número de elementos y nombre del arreglo.

Un arreglo consta de posiciones de memoria contiguas. La dirección más baja corresponde al primer elemento y la más alta al último. Para acceder a un elemento en particular se utiliza un índice.

En C, todos los arreglos usan cero como indice para el primer elemento y si el tamaño es n, el indice del último elemento es n-1.

Ejemplo de programa que utiliza un array:

```
main ()
{
      int arreglo2[TAMANIO];
```

```
/* cargo array con valor igual al indice mas 10*/
      for (int i=0;i<TAMANIO;i++)
          arreglo2[i]=i+10;

/* imprimo contenido del arreglo */
      for (int i=0;i<TAMANIO;i++)
          printf("Elemento %d del array es %d\n",i+1,arreglo2[i]);
}
```

Del programa anterior podemos extraer que :

1. para cargar un elemento de un array coloco el nombre de array seguido de un indice entre parentesis rectos y asocio el valor que desee.
2. para acceder al valor de un elemento del array coloco el nombre de array seguido de un indice entre parentesis rectos.
3. los indices con los que accedo al array varían entre 0 y la cantidad de elementos menos 1.

Los nombres de arrays siguen la misma convención que los nombres de variable.

Ejemplos de declaraciones:

```
int notas[8] /* almacena ocho notas */

char nombre[21] /* almacena nombres de largo menor o igual a 20 */

int multiplos[n] /* donde n tiene un valor, declara un arreglo de tamaño n*/
```

Los indices pueden ser cualquier expresión entera. Si un programa utiliza una expresión como subindice esta se evalua para determinar el índice. Por ejemplo si a=5 y b=10, el enunciado arreglo2[a+b] +=2, suma 2 al elemento del arreglo número 15.

Puedo utilizar un elemento del arreglo en las mismas expresiones que variables del tipo correspondiente. En el caso de arreglo2, puedo utilizar cualquiera de sus elementos en expresiones donde pueda utilizar una variable entera, por ejemplo

```
printf(" %d", arreglo2[0]+arreglo2[15]+arreglo2[30]);
```

Los arreglos pueden ser declarados para que contengan distintos tipos de datos. Por ejemplo un arreglo del tipo char puede ser utilizado para almacenar una cadena de caracteres.

1.2. Inicialización

Los elementos de un arreglo pueden ser inicializados en la declaración del arreglo haciendo seguir a la declaración un signo de igual y una lista entre llaves de valores separados por comas.

Por ejemplo

int n[10]={32, 27, 64, 18, 95, 24, 90, 70, 8, 3};

Si en la declaración hay menos inicializadores que el tamaño del array, los elementos son inicializados a cero. Puedo entonces inicializar todo un array en 0 con la declaración:

int n[10]={0};

Declarar más inicializadores que el tamaño del arreglo es un error de sintaxis.

Si en una declaración con una lista inicializadora se omite el tamaño del arreglo el número de elementos del arreglo será el número de elementos incluidos en la lista inicializadora.

Por ejemplo

int a1 [] = {1,2,3,4,5};

crea un arreglo de 5 elementos.

No se puede asignar un arreglo en otro, se tiene que copiar posición a posición.

1.3. Operaciones frecuentes:

Recorrido de un arreglo de tamaño n:

- El índice puede ir del primero al último elemento:

```
for (i=0;i < n;i++)
    {
     proceso
    }
```

- El índice puede ir del último al primer elemento:

```
for (i=n-1;i >= 0;i–)
    {
     proceso
    }
```

1.4. Cómo pasar arreglos a funciones, ejemplos: funciones en arreglos de enteros

Supongamos quiero definir funciones para modularizar el manejo de arreglos de enteros. Definiremos funciones que leen datos y los almacenan en arreglos, definiremos funciones que imprimen los datos almacenados en un arreglo, definiremos una función que halla el máximo en un arreglo, definiremos una función que suma los elementos en un arreglo y usando ésta última una función que calcula el promedio de un arreglo.

```
/* Funcion leo_arreglo, lectura de un arreglo de enteros */
void leo_arreglo(int a[],int n)
{
      for (int i=0;i < n;i++)
          {
           printf("Ingrese elemento :");
           scanf ("%d",&a[i]);
           printf("\n");
          }
}
```

invocamos la función anterior pasando como argumento el nombre del arreglo y su tamaño, por ejemplo:

leo_arreglo(array2,TAMANIO)

Estudiemos la función. Recibe como parametros un arreglo (parámetro a) y un entero (parámetro n) que representa el tamaño del arreglo.

C pasa de forma automática los arreglos a las funciones utilizando simulación de llamadas por referencia (cualquier modificación que realice en la función al array tendra efecto en el array que se pasa como parámetro). Esto es asi porque el nombre del arreglo coincide con la dirección del primer elemento del arreglo.

A pesar de que se pasan arreglos completos simulando llamadas por referencia, los elementos individuales de arreglo se pasan en llamadas por valor.

Para que una función reciba un arreglo en una llamada de función la lista de parametros de función debe especificar que se va a recibir un arreglo (lo hacemos colocando tipo, nombre y los parentesis rectos). No es obligatorio (pero es útil) pasar como argumento el tamaño del array.

Si se indica el tamaño del arreglo dentro de los paréntesis rectos el compilador lo ignorará.

Como el pasaje del array es por referencia, cuando la función utiliza el nombre del arreglo **a** se estará refiriendo al arreglo real en el llamador.

Veamos las otras funciones:

```
/* Funcion imprimo_arreglo, impresion de un arreglo de enteros */
```

```
void imprimo_arreglo(int a[],int n)
{
      for (int i=0;i<n;i++)
          {
           printf("Elemento numero %d = %d", i+1, a[i]);
           printf("\n");
          }
}
```

```
/* funcion buscar_maximo: halla el maximo en un arreglo */
int buscar_maximo(double valores[], int n)
{
    int maximo_pos = 0;
    for (int i = 1; i < n; i++) {
        if (valores[i] > valores[maximo_pos]) {
          maximo_pos = i;
        }
    }
    return maximo_pos;
}
```

```
/* Funcion sumo_arreglo, devuelve la suma de un arreglo de enteros */
int sumo_arreglo(int a[],int n)
{
    int suma=0;

    for (int i=0;i<n;i++)
        suma += a[i];
    return suma;
}
```

```
/* Funcion promedio: devuelve el promedio de un arreglo */
float promedio(int a[],int n)
{
      return sumo_arreglo(a,n)/n;
}
```

1.5. Ejemplos

1. De funciones y programas que computan arreglos:

   - Función que inicializa un arreglo de n lugares con los valores del 1 al n:

     void inicializo(int a[],int n)

```
{
      for(int i=0;i<n;i++) a[i]=i+1;
}
```

- Función que calcula la suma de dos arreglos de n elementos reales:

```
void sumo(float a[],float b[],float c[],int n)
{
      for (int i=0;i <n ; i++)
          c[i]=a[i]+b[i];
}
```

- Función que calcule los primeros n números de Fibonacci:

```
void Fibonacci(int a[],int n)
{
      a[0]=0;
      a[1]=1;
      for (int i=2;i<n;i++)
          a[i]=a[i-1]+a[i-2];
}
```

- Función que invierte un arreglo :

```
void invierto(int b[],int n)
{
      int temp;

      for (int i=0;i < n/2;i++)
          {
            temp = b[i];
            b[i]=b[n-i-1];
            b[n-i-1] = temp;
          }
}
```

De otra manera:

```
void invierto(int b[],int n)
{
      int i,j,temp;

      for (i=0,j=n;i < j;i++,j–)
          {
            temp = b[i];
            b[i]=b[j];
            b[j] = temp;
          }
}
```

- Programa que cuenta número de dígitos, espacios y otros.

```
void main()
{
      int c, i, nblancos, notros;
      int ndigitos[10];

      nblancos = notros = 0;
      for (i = 0; i < 10; ++i) ndigitos[i] = 0;

      while ((c = getchar()) != EOF)
            if (c >= '0' && c <= '9')
              ++ndigitos[c - '0'];
            else if (c == ' ' || c == '\n' || c == '\t')
                  ++nblancos;
                else
                     ++notros;

      printf("digitos = ");
      for (i = 0; i < 10; ++i) printf(" %d ", ndigitos[i]);
      printf("blancos = %d, otros = %d\n", nblancos, notros);
      system("PAUSE");
}
```

- Programa que cuenta número de vocales y otros en la entrada.

```
main ()
{
      int c, i, notros;
      int nvocales[6];

      notros = 0;
      for (i = 0; i < 5; ++i) nvocales[i] = 0;

      while ((c = getchar()) != EOF)
            switch (c) {
                  case 'a' : case 'A': ++nvocales[0];break;
                  case 'e' : case 'E': ++nvocales[1];break;
                  case 'i' : case 'I': ++nvocales[2];break;
                  case 'o' : case 'O': ++nvocales[3];break;
                  case 'u' : case 'U': ++nvocales[4];break;
                  default: ++notros; }

      printf("vocales = ");
      for (i = 0; i < 5; ++i)
          printf(" %d", nvocales[i]);
      printf("otros = %d\n", notros);
      system("PAUSE");
}
```

2. De uso de funciones sobre arreglos:

   - Programa que lee e imprime notas finales de 10 estudiantes (utilizando las funciones anteriores) :

```
main ()
{
        int notas[10];

        printf("Ingrese las notas finales de los 10 estudiantes\n");
        leo_arreglo(notas,10);
        printf("Las notas de los estudiantes son: \n");
        imprimo_arreglo(notas,10);
        system("PAUSE");
```

   - Programa que lee notas finales de 10 estudiantes e imprime su promedio (utilizando las funciones anteriores) :

```
main ()
{
        int notas[10];

        printf("Ingrese las notas finales de los 10 estudiantes\n");
        leo_arreglo(notas,10);
        printf("El promedio de las notas de los estudiantes es: %f", promedio(notas,10));
        system("PAUSE");
```

   - Programa que lee un valor n, almacena los primeros n números de Fibonacci y los imprime:

```
main ()
{
        int n;

        printf("Ingrese numero :");
        scanf(" %d", &n);
        putchar('\n');
        int fib[n];

        Fibonacci(fib,n);
        printf("Primeros %d numeros de Fibonacci son :\n");
        imprimo_arreglo(fib,n);
        system("PAUSE");
```

1.6. Arreglos constantes

Podrı'an existir situaciones en las cuales no queremos permitir que una función modifique un arreglo.

C proporciona un calificador llamado **const** que se puede utilizar para evitar la modificación de variables en particular de arreglos.

Cuando en un parametro de una función que representa un arreglo se antecede la declaración del parametro con la palabra clave **const**, los elementos del arreglo se convierten en constantes y cualquier intento de modificar un elemento del arreglo dentro de la función da como resultado un error en tiempo de compilación.

Ejemplo:

```
void trato_de_modificar(const int b [])
{
     b[0] = 1;
}
```

dará error de sintaxis.

1.7. Arreglos de caracteres o strings

Hasta ahora solo nos hemos ocupado de arreglos de enteros. Sin embargo los arreglos son capaces de contener datos de cualquier tipo. Estudiaremos el almacenamiento de **strings** o **cadenas** en arreglos de caracteres.

Un arreglo de caracteres puede ser inicializado utilizando una cadena, por ejemplo, la siguiente declaración:

```
char string1[] = "primero";
```

inicializa los elementos del arreglo string1 con las letras de la palabra primero. El tamaño del arreglo queda determinado por la cantidad de letras de primero.

Es importante notar que la cadena "primeroçontiene 7 caracteres más un caracter especial que indica la terminación de la cadena que es el '\0' o caracter nulo. Entonces primero realmente consta de 8 caracteres y este es el tamaño del array string1.

La declaración anterior es equivalente a :

```
char string1 [] = {'p', 'r', 'i', 'm', 'e', 'r', 'o', '\0'}
```

Dado que una cadena es un arreglo de caracteres podemos tener acceso a los caracteres individuales de una cadena utilizando la notación de nombre de arreglos con subindices.

1.7.1. Lectura e impresion de cadenas

Podemos imprimir y leer una cadena con el especificador de conversión " %s".

Los enunciados

```
printf(" %s","Hola");
printf(" %s",string1);
```

imprimiran Hola y primero. El enunciado:

```
char string2[20];

scanf(" %s", string2);
```

lee una cadena del teclado y la coloca en string2. Notar que el nombre del arreglo se pasa a scanf sin colocar el & que en otros casos se usa. El & es utilizado por lo regular para darle a scanf la localización de una variable en la memoria. Como el nombre de un arreglo es la dirección de inicio del arreglo el & no es necesario.

Es responsabilidad del programador asegurarse que el arreglo al cual se lee la cadena sea capaz de contener cualquier cadena que el usuario escriba en el teclado. La función scanf lee caracteres del teclado hasta encontrarse con el primer caracter de espacio en blanco. Se almacenará la cadena ingresada y el caracter nulo.

printf imprime hasta que encuentra el caracter nulo.

Si leemos una cadena con getchar(), es nuestra responsabilidad colocar el '\0' al final del arreglo.

Ejemplo:

```
/* leo cadena con getchar() */
main ()
{
      char ca[15];
      int i=0;
      while ((ca[i]=getchar())!=EOF) i++ ;
      ca[i]='\0';
}
```

1.8. Casos de Estudio, funciones con arreglos de caracteres

Ejemplo 1

Escribamos un programa que lee un conjunto de lineas e imprime la más larga. Un esquema del programa es:

```
mientras haya otra linea
        si es mas larga que la anterior mas larga
           salvar la linea y su largo
imprimir linea mas larga
```

Este esquema muestra que el programa se puede dividir naturalmente en funciones. Una pieza lee una nueva linea, main testea si es más larga, otra función salva la linea y main realiza el resto.

Empezemos escribiendo una función **getline** que lee una linea de la entrada. Esta función lee hasta que encuentra EOF o nueva linea.Retorna el largo de la linea leida y coloca el '\0' al final del string leido. Como el EOF se ingresa en una linea nueva, cuando se ingrese EOF se devolverá largo cero y el string consistirá solo en el '\0'.

Para salvar la linea utilizamos la función copiar.

Las funciones son las siguientes:

```
/* funcion getline */
int getline(char s[],int n)
{
    int c, i=0;

    while (--n > 0 && (c=getchar())!=EOF && c!='\n')
          s[i++] = c;
    if (c == '\n') s[i++]=c;
    s[i]='\0';
    return i;
}


/* funcion copiar, reliza una copia del array que se ingresa */
void copiar(char s1[], char s2[])
{
     int i=0;

     while ((s2[i]=s1[i])!='\0') i++;
}
```

la función main es:

```
#define MAX 1000

main ()
{
      int largo,maximo=0;
      char linea[MAX];
      char copia[MAX];

      while ((largo=getline(linea,MAX))>0)
            if (largo > maximo)
               {
                maximo=largo;
                copiar(linea,copia);
               }
      if (maximo > 0) printf(" %s", copia); }
```

Ejemplo 2

Veamos un programa que lee un conjunto de lineas e imprime cada linea de la entrada que contiene un patron particular. Por ejemplo

Es el tiempo
para todo el bien
el hombre va en busca
de nuevas sensaciones.

Si buscamos el patrón **el**, lo encontramos en las primeras tres lineas.

La estructura básica del programa que resuelve el problema es:

```
mientras haya otra linea
        si la linea contiene el patron
            imprimo la linea
```

Si bien es posible poner todo el código del programa en la instrucción main, una manera mejor de definir el programa es hacer de cada operación una función separada.

Utilizaremos la función **getline** del ejemplo anterior.

Definiremos una función **indice** que regresa la posición en la linea de entrada en la cual se encontró el patrón, retorna -1 si el patrón no se encontró.

Por último main imprime cada linea que contiene el patrón.

```
/* funcion indice, retorna indice de t en s, -1 si no se encuentra */
int indice(char s[], char t[],int n)
{
    int i,j,k;

    for (i=0;s[i]!='\0';i++)
        {
          for (j=i,k=0; t[k]!='\0' && s[j]==t[k]; j++, k++) ;
          if (t[k] == '\0') return i;
        }
    return -1;
}
```

la función main que llama estas funciones es la siguiente:

```
#define MAX 1000
#define largo 10
main ()
{
       char linea[MAX];
       char patron[largo];

       printf("Ingrese patron");
       scanf(" %s", patron);
       while (getline(linea,MAX) > 0)
```

```
            if (indice(linea,patron) >= 0)
              printf(" %s",linea);
}
```

2. Arreglos con múltiples subindices

En C los arreglos pueden tener múltiples subindices. Podemos utilizarlos por ejemplo en tablas de valores que almacena información arreglada en filas y columnas. Para identificar un elemento de la tabla debemos especificar dos subindices, el primero especifica el renglón del argumento y el segundo identifica la columna.

Podemos especificar arreglos con más de 2 subindices.

Un arreglo de multiple subindice puede ser inicializado en su declaración en forma similar a los de un subindice. Por ejemplo un arreglo de doble subindice puede ser declarado e inicializado con:

```
int b[2][3] = { {1,2,3}, {3,4,5}}
```

si para un renglón dado no se proporcionan suficientes inicializadores los elementos restantes de dicho inicializador se setarán en cero.

Veamos una función imprimo_arreglo que imprima los valores de un array de dos dimensiones por fila.

```
/* imprimo array de dos dimensiones por fila */

void imprimo_arreglo(int a [][3],int n)
{
    int i,j;

    for(i=0;i < n;i++)
    {
       printf("Fila %d\n", i);
       for (j=0;j < 3;j++)
           printf(" %d ", a[i][j]);
       printf("\n");
    }
}
```

Notar que la definición de la función especifica el parámetro del arreglo como int a[][3]. El primer subindice de un arreglo de multiples subindices no se requiere pero todos los demás subindices son requeridos. El compilador utiliza estos subindices para determinar las localizaciones en memoria de los elementos en los arreglos de multiples subindices. Los elementos del arreglo son almacenados en memoria de forma consecutiva renglón despues de renglón.

Si queremos generalizar el programa, podemos definir la cantidad de columnas en una constante del siguiente modo:

```
/* imprimo array de dos dimensiones por fila */

#define nro_columnas 3

void imprimo_arreglo(int a [][nro_columnas],int n)
{
      int i,j;

      for(i=0;i < n;i++)
      {
         printf("Fila %d\n", i);
         for (j=0;j < nro_columnas;j++)
             printf(" %d ", a[i][j]);
         printf("\n");
      }
}
```

Ejemplo

Consideremos un arreglo donde cada renglon representa un alumno y cada columna representa una calificación en uno de los cuatro examenes que los alumnos pasaron durante el semestre.

Utilizaremos las siguientes funciones: **minimo** determina la calificación más baja de entre las calificaciones de todos los estudiantes. **maximo** determina la calificación más alta de entre las calificaciones de todos los estudiantes. **promedio** determina el promedio para el semestre de un alumno en particular. **imprimo_arreglo** es la función que ya vimos.

Las funciones minimo y maximo reciben tres argumentos: el arreglo llamado **notas**, el número de alumnos (renglones) y el número de examenes (columnas). Las funciones iteran a través del arreglo notas utilizando estructuras for anidadas.

Las funciones se presentan a continuación:

```
/* funcion minimo */
int minimo(int notas[][nro_examenes],int n,int m)
{
    int i,j,menorgrado=notas[0][0];

    for (i=0; i < n;i++)
        for (j=0; j < m; j++)
            if (notas[i][j] < menorgrado) menorgrado = notas[i][j];
    return menorgrado;
}


/* funcion maximo */
```

```
int maximo(int notas[][nro_examenes],int n,int m)
{
    int i,j,mayorgrado=notas[0][0];

    for (i=0; i < n;i++)
        for (j=0; j < m; j++)
            if (notas[i][j] > mayorgrado) mayorgrado = notas[i][j];
    return mayorgrado;
}


/* funcion promedio */
int promedio(int notas_de_alumno [],int n)
{
    int i,total=0;

    for (i=0; i < n;i++)
        total += notas_de_alumno[i];
    return (float) total/n;
}
```

la función promedio se invoca con una fila del arreglo notas del siguiente modo:

```
promedio(notas[alumno],cant_examenes);
```

Principios de Programación

El lenguaje C

1. Estructuras

Las estructuras son colecciones de variables relacionadas bajo un nombre. Las estructuras pueden contener variables de muchos tipos diferentes de datos - a diferencia de los arreglos que contienen unicamente elementos de un mismo tipo de datos.

1.1. Definicion de estructuras

Las estructuras son *tipos de datos derivados* - están construidas utilizando objetos de otros tipos. Considere la siguiente definición de estructura:

```
struct ejemplo {
        char c;
        int i;};
```

La palabra reservada **struct** indica se está definiendo una estructura. El identificador ejemplo es el nombre de la estructura. Las variables declaradas dentro de las llaves de la definición de estructura son los *miembros* de la estructura. Los miembros de la misma estructura deben tener nombres únicos mientras que dos estructuras diferentes pueden tener miembros con el mismo nombre. Cada definición de estructura debe terminar con un punto y coma.

La definición de struct ejemplo contiene un miembro de tipo char y otro de tipo int. Los miembros de una estructura pueden ser variables de los tipos de datos básicos (int, char, float,etc) o agregados como ser arreglos y otras estructuras. Una estructura no puede contener una instancia de si misma.

Declaramos variables del tipo estructura del siguiente modo:

```
struct ejemplo e1, a[10];
```

o alternativamente sin usar la palabra struct:

```
ejemplo e1, a[10];
```

Las declaraciones anteriores declaran variables e1 de tipo ejemplo y a de tipo arreglo de ejemplo de dimensión 10.

Se pueden declarar variables de tipo estructura ejemplo colocando sus nombres a continuación de la llave de cierre de la definición de estructura y el punto y coma, en el caso anterior:

```
struct ejemplo {
        char c;
        int i;} e1, a[10];
```

Una operación válida entre estructuras es asignar variables de estructura a variables de estructura del mismo tipo. Las estructuras no pueden compararse entre si.

Ejemplo

Consideremos la información de una fecha. Una fecha consiste de: el día, el mes, el año y posiblemente el día en el año y el nombre del mes. Declaramos toda esa información en una estructura del siguiente modo:

```
struct fecha {
        int dia;
        int mes;
        int anio;
        int dia_del_anio;
        char nombre_mes[9];
};
```

Un ejemplo de estructura que incluye otras estructuras es la siguiente estructura persona que incluye la estructura fecha:

```
struct persona {
        char nombre[tamanio_nombre];
        char direccion[tamanio_dir];
        long codigo_postal;
        long seguridad_social;
        double salario;
        fecha cumpleanios;
        fecha contrato;
};
```

1.2. Cómo inicializar estructuras

Las estructuras pueden ser inicializadas mediante listas de inicialización como con los arreglos. Para inicializar una estructura escriba en la declaración de la variable a continuación del nombre de la variable un signo igual con los inicializadores entre llaves y separados por coma por ejemplo:

```
ejemplo e1 = { 'a', 10 };
```

Si en la lista aparecen menos inicializadores que en la estructura los miembros restantes son automaticamente inicializados a 0.

Las variables de estructura también pueden ser inicializadas en enunciados de asignación asignandoles una variable del mismo tipo o asignandole valores a los miembros individuales de la estructura.

Podemos inicializar una variable del tipo fecha como sigue:

```
struct fecha f = {4, 7, 1776, 186, "Julio"};
```

1.3. Cómo tener acceso a los miembros de estructuras

Para tener acceso a miembros de estructuras utilizamos el operador punto. El operador punto se utiliza colocando el nombre de la variable de tipo estructura seguido de un punto y seguido del nombre del miembro de la estructura. Por ejemplo, para imprimir el miembro c de tipo char de la estructura e1 utilizamos el enunciado:

printf (" %c, e1.c);

Para acceder al miembro i de la estructura e1 escribimos: e1.i

En general, un miembro de una estructura particular es referenciada por una construcción de la forma:

nombre_de_estructura.miembro

por ejemplo para chequear el nombre de mes podemos utilizar:

if (strcmp(d.nombre_mes,"Agosto")==0) .....

1.4. Cómo utilizar estructuras con funciones

Las estructuras pueden ser pasadas a funciones pasando miembros de estructura individuales o pasando toda la estructura.

Cuando se pasan estructuras o miembros individuales de estructura a una función se pasan por llamada por valor. Para pasar una estructura en llamada por referencia tenemos que colocar el '*' o '&'.

Los arreglos de estructura como todos los demás arreglos son automaticamente pasados en llamadas por referencia.

Si quisieramos pasar un arreglo en llamada por valor, podemos definir una estructura con único miembro el array.

Una función puede devolver una estructura como valor.

Ejemplo

Consideraremos el ejemplo de un punto dado por dos coordenadas enteras.

```
struct punto {
        int x;
        int y;};

/* creo_punto: crea un punto a partir de sus coordenadas */
punto creo_punto(int a, int b)
{
        punto temp;

        temp.x=a;
        temp.y=b;
        return temp;
}
```

```
/* sumo_puntos: suma dos puntos */
punto sumo_puntos(punto p1,punto p2)
{
        p1.x += p2.x;
        p1.y += p2.y;
        return p1;
}

/* imprimo_punto: imprime las coordenadas de un punto */
void imprimo_punto(punto p)
{
      printf("Coordenadas del punto: %d y %d \n", p.x, p.y);
}
```

1.5. Typedef

La palabra reservada **typedef** proporciona un mecanismo para la creación de sinónimos (o alias) para tipos de datos anteriormente definidos. Por ejemplo:

```
typedef struct ejemplo Ejemplo;
```

define Ejemplo como un sinónimo de ejemplo.
Una forma alternativa de definir una estrutura es:

```
typedef struct {
        char c;
        int i;} Ejemplo;
```

Podemos ahora utilizar Ejemplo para declarar variables del tipo struct, por ejemplo

```
Ejemplo a[10];
```

typedef se utiliza a menudo para crear seudónimos para los tipos de datos básicos. Si tenemos por ejemplo un programa que requiere enteros de 4 bytes podría usar el tipo **int** en un programa y el tipo **long** en otro. Para garantizar portabilidad podemos utilizar typedef para crear un alias de los enteros de 4 bytes en ambos sistemas.

1.6. Ejemplo: simulación de barajar y distribuir cartas

El programa que sigue se basa en la simulación de barajar y distribuir cartas. El programa representa el mazo de cartas como un arreglo de estructuras, donde cada estructura contiene el número de la carta y el palo.

Las funciones son: **inicializar_mazo** que inicializa un array de cartas con los valores de las cartas ordenado del 1 al 12 de cada uno de los palos, **barajar** recibe un array de 48 cartas y para cada una de ellas se toma un número al

azar entre el 0 y el 47. A continuación se intercambia la carta original con la seleccionada al azar. Finalmente la función imprimir imprime las cartas. Se utiliza para imprimir las cartas luego de barajar.

La función copiar es auxiliar dentro de inicializar_mazo, es necesaria porque ISO C++ prohibe la asignación de arrays (ISO=International Organization for Standardization, red formada por distintos paises, no gubernamental. En algunos paises sus miembros son parte de la estructura gubernamental y en otros son parte del sector privado. En Uruguay depende del gobierno desde el 97).

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

struct carta {
        int numero;
        char palo[7];};

typedef carta Carta;
typedef char Palo[7];

void inicializar_mazo(Carta m[],Palo p[]);

void barajar(Carta m[]);

void imprimo(Carta m[]);

main ()
{
       Carta mazo[48];
       Palo p[4] = {"copa","oro","espada","basto"};

       srand(time(NULL));
       inicializar_mazo(mazo,p);
       barajar(mazo);
       imprimir(mazo);
       system("PAUSE");
}

void copiar(char a, char b, int largo)
{
      int i;

      for (i=0;i < largo;i++)
          a[i]=b[i];
}

void inicializar_mazo(Carta m[],Palo p[])
{
      int i;
```

```
        for (i=0; i < 48;i++)
        {
            m[i].numero=(i % 12)+1;
            copiar(m[i].palo,p[i/12],7);
        }
}

void barajar(Carta m[])
{
        int i,j;
        Carta temp;

        for (i=0; i < 48; i++)
        {
            j = rand() % 48;
            temp = m[i];
            m[i] = m[j];
            m[j] = temp;
        }
}

void imprimir(Carta m[])
{
        int i,j;
        char c;
        for (i=0; i < 48; i++)
        {
            printf (" %i de ", m[i].numero);
            printf (" %s ", m[i].palo);
            printf ("\n");
        }
}
```

Principios de Programación

El lenguaje C

1. Variables locales y globales

1.1. Variables locales

Las funciones permiten al programador modularizar un programa. Todas las variables declaradas en las definiciones de función son *variables locales* - son conocidas sólo en la función en la cual están definidas (ninguna otra función tiene acceso a ellas). La mayor parte de las funciones tienen una lista de *parámetros*. Los parámetros de una función también son variables locales. Las variables locales comienzan su existencia cuando la función es llamada y desaparecen cuando la función termina su ejecución, por esta razón se conocen como *variables automáticas*. Las variables automáticas no retienen sus valores de una llamada a otra.

1.2. Variables globales

Es posible definir funciones que son *externas* a todas las funciones, esto es variables globales que pueden ser accedidas por cualquier función. Pueden ser utilizadas en el lugar de las listas de parámetros para comunicar información entre funciones. Las variables externas existen permanentemente y retienen su valores aun despues de que las funciones que las setean han terminado su ejecución.

Las variables externas deben ser definidas exactamente una vez, fuera de todas las funciones, esto les asigna almacenamiento. La variable debe también ser declarada en cada función que quiera acceder a ella. La declaración debe ser una sentencia *extern*. Veamos un ejemplo:

```
#include <stdio.h>

#define MAX_LINEA 1000 /* maximo tamanio de linea de entrada */

int max; /* maximo tamanio visto hasta el momento */
char linea[MAX_LINEA] /* linea actual */
char mas_larga[MAX_LINEA] /* linea mas larga hasta el momento */

int getline(void);
void copy(void);

/* imprimir linea de la entrada mas larga */
main ()
{
      int len;
      extern int max;
      extern char mas_larga[];

      max = 0;
```

```
        while ((len = getline()) ¿0)
                if (len ¿max)
                {
                  max=len;
                  copio();
                }
        if (max > 0) printf(" %s", mas_larga);
        system("PAUSE");
}

/* getline: lee una linea de la entrada y devuelve su largo */
int getline(void)
{
    int c,i;
    extern char linea[];

    for (i=0;i < MAX_LINEA - 1 && (c=getchar())!EOF && c != '\n';++i)
        linea[i]=c;
    if (c=='\n') {
      linea[i]=c;
      ++i; }
    linea[i]='\0';
    return i;
}

/* copio: copia una linea en otra */
void copio(void)
{
      int i;
      extern char linea[], mas_larga[];

      i=0;
      while ((mas_larga[i]=linea[i]) != '\0') i++;
}
```

En ciertas circunstancias las declaraciones *extern* pueden omitirse: si la definición de una variable externa aparece en el archivo antes de su uso en una función particular no hay necesidad de declaración *extern* en la función.

2. Reglas de alcance

El alcance de un nombre es la parte del programa en la cual el nombre está definido.

Para una variable automática declarada en una función el alcance es la función en la cual el nombre está declarado y variables con el mismo nombre en distintas funciones no están relacionadas. Lo mismo es cierto para los argumentos de la función.

El alcance de una variable externa es desde el punto en el cual es definida en un archivo hasta el final del archivo. Si necesitamos referenciar una variable

externa antes de ser definida o si es utilizada en un archivo distinto es necesario colocar la declaración extern.

3. Clases de almacenamiento

La clase de almacenamiento de un identificador ayuda a determinar su duración de almacenamiento y su alcance. La duración de almacenamiento de un identificador es el periodo durante el cual dicho identificador existe en memoria. El alcance de un identificador en un programa es donde puede ser referenciado.

3.1. Variables Estáticas

Existen dos tipos de identificadores con persistencia estática: los identificadores externos y las variables locales declaradas como **static**.

Pueden ser internas o externas. Variables estáticas internas son locales a la función en la que se definen pero se diferencian en que continuan en existencia entre una llamada y otra de la función. Estas variables conservan el valor con el que salen de la función. Luego, las variables internas estáticas tienen almacenamiento permanente en la función.

Una variable estática externa es conocida en el resto del archivo en el cual es declarada pero no en ningún otro archivo.

Variables estáticas se declaran prefijando a la declaración la palabra **static**.

Es posible declarar funciones como estáticas, esto hace que su nombre sea desconocido fuera del archivo en el cual es declarada.

3.2. Variables registro

Los datos de un programa en la versión en lenguaje de máquina para cálculos y otros procesos normalmente se cargan en registros.

Una declaración **register** le avisa al compilador que la variable en cuestión va a ser utilizada ampliamente. Cuando es posible, variables registro se colocan en los registros de la máquina lo cual va a resultar es programas más rapidos.

Las declaraciones son de la forma:

```
register int x;
register char c;
```

las variables registro se pueden aplicar solo a variables automáticas y a los parámetros formales de una función.

El compilador puede ignorar declaraciones register. Puede que no exista un número suficiente de registros disponibles.

4. Más sobre reglas de alcance

Hay cuatro alcances posibles:

1. alcance de función
2. alcance de archivo

3. alcance de bloque

4. alcance de prototipo de función

4.1. Alcance de función

Las etiquetas (identificador seguido por dos puntos) son los únicos identificadores con *alcance de función*. Pueden ser utilizadas en cualquier parte dentro de la función en la cual aparecen, pero no pueden ser utilizadas fuera del cuerpo de la función. Se utilizan en estructuras switch (como etiquetas case) y en **goto**.

4.2. Alcance de archivo

Un identificador declarado por fuera de cualquier función tiene alcance de archivo. Es conocido en todas las funciones desde el punto donde el identificador se declara hasta el final del archivo. Las variables globales, las definiciones de función y los prototipos de función colocados fuera de una función tienen alcance de archivo.

4.3. Alcance de bloque

Los identificadores dentro de un bloque tienen alcance de bloque. Este termina en la llave derecha de terminación del bloque. Las variables locales declaradas al principio de una función tienen alcance de bloque al igual que los parámetros de la función. Cuando los bloques están anidados y un identificador del bloque externo tiene el mismo nombre que un identificador del bloque interno, el identificador del bloque externo estará "oculto"hasta que el bloque interno termine.

Las variables locales declaradas static tendrán alcance de bloque aunque existan a partir del momento en que empieza a ejecutarse el programa.

4.4. Alcance del prototipo de función

Los únicos identificadores con este alcance son los que se declaran en la lista de parámetros del prototipo de una función. El compilador ignora estos nombres.

4.5. Ejemplo

```
/* Un ejemplo de alcance */
#include <stdio.h>

void a(void);
void b(void);
void c(void);

int x = 1; /* variable global */

main ()
{
      int x=5;
```

```
        printf("x local en alcance exterior de main es %d\n",x);
        { /* nuevo alcance */
          int x=7;
          printf("x local en alcance interior de main es %d\n",x);
        }
        printf("x local en alcance exterior de main es %d\n",x);
        a();
        b();
        c();
        a();
        b();
        c();
        printf("x local en main es %d\n",x);
        system("PAUSE");
}

void a(void)
{
      int x = 25;
      printf("\n x local en a es %d luego de entrar a a\n",x);
      ++x;
      printf("x local en a es %d antes de salir de a\n",x);
}

void b(void)
{
      static int x=50;
      printf("\n x local static en b es %d luego de entrar a b\n",x);
      ++x;
      printf("x local static en b es %d antes de salir de b\n",x);
}

void c(void)
      printf("\n x global es %d al entrar a c\n",x);
      x*=10;
      printf("x global es %d antes de salir de c\n",x);
}
```

La salida del programa es:

```
x local en alcance exterior de main es 5
x local en alcance interior de main es 7
x local en alcance exterior de main es 5

x local en a es 25 luego de entrar a a
x local en a es 26 antes de salir de a

x local static en b es 50 luego de entrar a b
x local static en b es 51 antes de salir de b
```

x global es 1 al entrar a c
x global es 10 antes de salir de c

x local en a es 25 luego de entrar a a
x local en a es 26 antes de salir de a

x local static en b es 51 luego de entrar a b
x local static en b es 52 antes de salir de b

x global es 10 al entrar a c
x global es 100 antes de salir de c
x local en main es 5

Principios de Programación

El lenguaje C

1. Funciones Recursivas

Para algunos tipos de problemas es útil tener funciones que se llaman a si mismas. Una *función recursiva* es una función que se llama a si misma.

Primero consideraremos la recusión en forma conceptual y a continuación examinaremos varios programas que contienen funciones recursivas.

Las funciones recursivas se definen definiendo:

1. caso base: son casos simples.Si la función es llamada con el caso base la función simplemente devuelve un resultado.

2. caso recursivo: la función es llamada con un problema más complejo. Para hacer factible la recursión este último debe parecerse al problema original. El paso de recursión puede dar como resultado muchas llamadas recursivas a la función con problemas más sencillos que el original.

A fin de que la recursión en forma eventual se termine, cada vez que la función se llame a si misma debe ser sobre una versión ligeramente más sencilla que el problema original. Esta secuencia de problemas más pequeños deben de converger en el paso base.

Ejemplo 1: Factorial

El factorial de un entero no negativo, escrito n! y pronunciado factorial de n, es el producto:

n.(n-1).(n-2).. . . 1

con 1!=1 y 0!=1.

El factorial de un entero *numero* mayor o igual a 0 puede ser calculado en forma iterativa utilizando for como sigue:

```
factorial=1;
for (cont=numero;cont >= 1;cont - - )
    factorial *= cont;
```

Una definición recursiva de la función factorial se obtiene al observar la siguiente relación:

n!=n.(n-1)!

la definición de la función recursiva correspondiente a factorial es:

```
long factorial(int numero)
{
      if (numero <= 1) return 1;
      else return (numero * factorial(numero-1));
}
```

utilizamos variables de tipo long porque la función factorial crece rápido.

Ejemplo 2: Fibonacci

La serie de Fibonacci:

0,1,1,2,3,5,8,13,21,...

empieza con 0 y 1 y tiene la propiedad que cada número subsecuente es la suma de los dos números previos.

Puede ser definida en forma recursiva por:

fibonacci(0)=0
fibonacci(1)=1
fibonacci(n)=fibonacci(n-1)+fibonacci(n-2)

el programa respectivo es:

```
long fibonacci (int n)
{
      if (n==0 || n==1) return n;
      else return fibonacci(n-1)+fibonacci(n-2);
}
```

Cada vez que se invoca a fibonacci se prueba el caso base, es decir si n es 0 o 1. Si esto es verdadero se regresa n. Si n es mayor que 1 el paso de recursión genera dos llamadas recursivas cada una de las cuales es para un problema ligeramente más sencillo que el original (valores de n menores).

Ejemplo 3

Consideremos una función que recibe un número n e imprime los números del n al 1:

```
void mi_funcion(int cont)
{
      if (cont==0) return;
      else {
            printf(" %d ",cont);
            mi_funcion(–cont);
            return;}
```

observar que llamamos recursivamente con - - cont. Si pusieramos en cambio cont - -, la función entraria en loop ya que llamaria con cont sin restar.

Ejemplo 4: El triangulo de Pascal

Lo utilizamos para calcular las C_m^n.Este número cuenta la cantidad de formas de escoger m objetos de un total de n objetos distintos. Les llamamos *combinaciones* de m objetos en n.

El triangulo de Pascal se construye como sigue: al principio se coloca un 1 (que corresponde a C_0^0). Para cada renglón subsecuente, digamos para el renglón n, se coloca un 1 a la izquierda y un 1 a la derecha que corresponden con C_0^n y C_n^n respectivamente. Los elementos restantes se calculan sumando los dos números que tiene justo arriba a la izquierda y a la derecha, es decir $C_m^n = C_{m-1}^{n-1} + C_m^{n-1}$ para todo $0 < m < n$.

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
```

El programa para calcular combinaciones es el siguiente:

```
int comb(int n,int m)
{
    if ((n==0) || (n==m)) return 1;
    else return comb(n-1,m-1) + comb(n-1,m);
}
```

Ejemplo 5: printd

Este ejemplo consiste en imprimir un número como un string de caracteres. Llamaremos recursivamente con el cociente de dividir entre 10. Por ejemplo, supongamos tenemos el número 3267. Llamaremos recursivamente con 326, con 32 y con 3. Cuando llegamos a 3 el cociente es 0 por lo cual no realizamos ninguna llamada recursiva. Luego de la llamada recursiva o si no hay llamada recursiva, imprimimos el módulo del número entre 10, por lo cual se imprimira: 3 (módulo de 3), 2 (módulo de 32), 6 (módulo de 326) y 7 (módulo de 3267).

El programa es el siguiente:

```
void printd (int n)
{
     int i;

     if (n ¡0)
     {
       putchar('-');
       n= -n;
     }
     if ((i=n/10)!=0) printd(i);
```

```
    putchar(n % 10 + '0');
}
```

2. Recursión en comparación con iteración

Compararemos los dos enfoques y analizaremos porqué el programador debe escoger un enfoque sobre el otro en una situación en particular.

Tanto la iteración como la recursión se basan en una estructura de control: la iteración utiliza una estructura de repetición, la recursión utiliza una estructura de selección. Tanto la iteración como la recursión implican repetición: la iteración utiliza la estructura de repetición en forma explicita, la recursión consigue la repetición mediante llamadas de función repetidas. La iteración y la recursión ambas involucran una prueba de terminación: la iteración termina cuando falla la condición de continuación del ciclo, la recursión termina cuando se reconoce un caso base.

La recursión tiene muchas negativas. La sobrecarga de llamadas de la función puede resultar costoso tanto en tiempo de procesador como en espacio de memoria. Cada llamada recursiva genera otra copia de la función, esto puede consumir gran cantidad de memoria. La iteración por lo regular ocurre dentro de una función por lo que no ocurre la sobrecarga de llamadas repetidas de función y asignación extra de memoria.

Principios de Programación

Algoritmos de Búsqueda y Ordenación

1. Algorimos de ordenación

Discutiremos el problema de ordenar un array de elementos. A los efectos de simplificar asumiremos que los arrays contienen solo enteros aunque obviamente estructuras más complicadas son posibles. Asumiremos también que el ordenamiento completo se puede realizar en memoria principal o sea la cantidad de elementos es relativamente pequeo (menos de un millón).

Ordenamientos que no se pueden realizar en memoria deben realizarse en disco. Se encuentran ejemplos en la bibliografía.

1.1. Preliminares

Los algoritmos que describiremos reciben como argumentos un array que pasa los elementos y un entero que representa la cantidad de elementos. Asumiremos que N (el número de elementos) es un número legal. Para alguno de los programas que veremos será conveniente utilizar un sentinela en posición 0, por lo cual nuestros array irán del 0 al N. Los datos irán del 1 al N.

Asumiremos la existencia de operadores de comparación $<$ y $>$. Llamaremos al ordenamiento que se basa en el uso de estos operadores *ordenamiento basado en comparaciones*.

1.2. Ordenamiento Burbuja

Consideremos el programa de ordenamiento **burbuja** que ordena un array de enteros en orden creciente:

```
void burbuja(int a[],int n)
{
        int i,j,temp;

        for(i=0;i<n-1;i++)
            for(j=0;j<n-1;j++)
                if (a[j]>a[j+1])
                  intercambio(&a[j],&a[j+1]);
}


void intercambio(int *a, int *b)
{
            int aux;
```

```
                aux=*a;
                *a = *b;
                *b=aux;
}
```

En cada ejecucion del for interior (en el que varia j) se coloca el elemento mayor en su lugar. Realizo esta repetición tantas veces como elementos.

1.3. Ordenamiento por inserción (Insertion Sort)

Es uno de los algoritmos más simples. Consiste en $N-1$ pasadas. En las pasadas 2 a N se cumplirá que los elementos de las posiciones 1 a P están ordenados. En la pasada P movemos el elemento P-esimo a su lugar correcto, este lugar es encontrado en las posiciones de los elementos 1 a P. El programa es el siguiente:

```
void Sort_por_insercion (int A[], int N)
{
        int j,P,tmp;
        {
            A[0]=Min_int;
            for(P=2;P<=N;P++)
                {
                 j=P;
                 tmp=A[P];
                 while(tmp < A[j-1])
                 {
                        A[j] = A[j-1];
                        j-=1;
                 }
                 A[j]=tmp;
                }
        }
}
```

La idea del programa es la siguiente: coloco un centinela en la primer posición (posición 0) por lo cual el elemento a insertar será colocado en caso de que sea el mínimo en la posición 1. Guardamos el valor del elemento a insertar en una variable auxiliar tmp. Si tmp es menor que A[j-1] su posición será anterior o igual a (j-1), luego corro el elemento de la posición (j-1) a la posición j para hacer lugar para tmp. Repito esto para todos los elementos anteriores al tmp. Cuando encuentre una posición tal qué tmp no es menor indica que los elementos anteriores están ordenados (estaban ordenados del 0 al $P-1$). Luego lo unico que resta es colocar tmp en la posición j.

1.4. Ordenamiento por Selección (Selection Sort)

La idea del selection sort es la siguiente : en la pasada i-esima seleccionamos el elemento menor entre $A[i], \ldots, A[n]$ y lo intercambiamos con el $A[i]$. Como

resultado, luego de i pasadas los menores i elementos ocuparán las posiciones $A[1], \ldots, A[i]$ y además los elementos de dichas posiciones estarán ordenados. El programa es el siguiente:

```
void Sort_por_seleccion (int A[], int N)
{
        int i,j,sel,clave_sel;
        {

            for(i=1;i<N;i++)
                {
                 sel=i;
                 clave_sel=A[i];
                 for(j=i+1;j<=N;j++)
                 {
                    if (A[j]<clave_sel)
                       { clave_sel=A[j];
                         sel=j;
                       }
                 }
                 intercambio(A[i],A[sel]);
                }
        }
}
```

2. Algoritmos de Búsqueda

Con frecuencia el programador trabajará con grandes cantidades de información almacenada en arreglos. Podría ser necesario determinar si algún arreglo contiene un valor que sea igual a cierto valor clave.

El proceso para encontrar un elemento particular en un arreglo se llama búsqueda. Estudiaremos dos técnicas de búsqueda: una técnica simple llamada busqueda lineal y una más eficiente llamada busqueda binaría.

Ambos programas se pueden implementar recursivamente o no. En este capítulo veremos la implementación no recursiva.

2.1. Búsqueda lineal

La búsqueda lineal compara los elementos del array con la clave de búsqueda hasta que encuentra el elemento o bien hasta que se determina que no se encuentra.

```
int busqueda_lineal (int A[], int clave, int n)
{
            for(int i=0;i<n;i++)
                if (A[i]==clave) return i;
```

```
        return -1;
}
```

Este método funciona bien con arreglos pequeos y con los no ordenados. En arreglos grandes u ordenados conviene aplicar la búsqueda binaria que es más eficiente.

2.2. Búsqueda binaria

Dados un entero X y un array A de n enteros que se encuentran ordenados y en memoria, encontrar un i talque $A[i] == X$ o retornar 0 si X no se encuentra en el array (consideramos los elementos del array de 1 a N.

La estrategia consiste en comparar X con el elemento del medio del array, si es igual entonces encontramos el elemento, sino, cuando X es menor que el elemento del medio aplicamos la misma estrategia al array a la izquierda del elemento del medio y si X es mayor que el elemento del medio aplicamos la misma estrategia al array a la derecha de dicho elemento.

Para simplificar el código definimos A[0]=X.

```
int busqueda_binaria(int A[],int X,int n)
{
        int izq=1,medio,der=n;

        A[0]=X;
        do
        {
            medio = (izq+der) / 2;
            if(izq > der)
              medio=0;
            else if A[medio] < X
                    izq = medio +1;
                 else der=medio-1;
        }
        while (A[medio] != X);
        return medio;
}
```