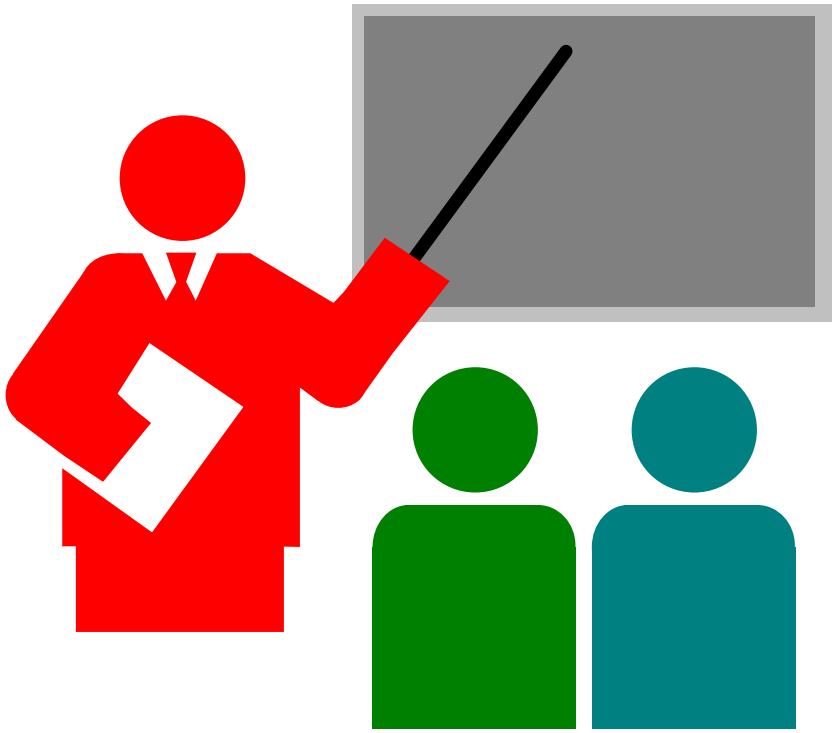
Manual para Usuario de Turbo C++



CAPITULO 1.- INTRODUCCION

CAPITULO 2.- TIPOS DE OPERADORES Y EXPRESIONES

CAPITULO 3.- CONTROL DE FLUJO

CAPITULO 4.- FUNCIONES Y ESTRUCTURA DE PROGRAMAS

CAPITULO 1 INTRODUCCION

Comenzaremos con una rápida introducción al lenguaje C. Mostraremos algunos elementos esenciales del lenguaje en programas reales, pero sin caer en grandes detalles, reglas formales y excepciones. En este sentido no trataremos de ser completos o aun precisos. Deseamos en este sentido que Ud. adquiera estos elementos tan rápidamente como sea posible, y nos concentraremos en lo básico : variables y constantes aritméticas, control de flujo, funciones, y las viejas instrucciones de I/O. Así, en efecto, intencionalmente dejamos fuera de este **CAPITULO** características de C que son de vital importancia
para escribir grandes programas. Esto incluye punteros, estructura; la mayoría de los lenguajes C son ricos en set de operadores, varias declaraciones de control de flujo, y también en innumerables detalles. Estas desventajas son aproximadas del curso, lo mas notable es que la historia completa de un lenguaje característico en particular no es fundado en un nico lugar.
En cualquier caso, programadores experimentados pueden ser capaces de extrapolar desde el material de este **CAPITULO** hasta sus propias necesidades de programación. Los principiantes pueden reforzarse escribiendo programas cortos.

1.1 ESTRUCTURA DE UN PROGRAMA EN C

El nico camino para aprender un nuevo lenguaje de programación es escribiendo o programando en este, esta es la barrera básica, al superarla Ud. será capaz de crear sus propios programas texto en alguna parte, compilarlos sucesivamente, cargarlos y correrlos. Cabe hacer notar que esto dependerá del sistema en que Ud. trabaje.

Como un primer ejemplo veamos el siguiente programa :

```
main ()
{
    printf (" hola , Freddy\n ");
}
```

En nuestro ejemplo main es como una función. Normalmente Ud. tendrá libertad para dar a cualquier función el nombre que guste, pero main es un nombre especial, sus programas comenzaran ejecutándose con la
expresión main; esto significa que todo programa debe tener algn main en alguna parte. main usualmente invocara otras funciones para ejecutar esta tarea .

Un método de comunicación de datos entre funciones, es por argumentos. El paréntesis seguido de un nombre de función rodea la lista de argumentos; aquí main es una función de argumentos, indicado por (). Las llaves encierran un conjunto de declaraciones.

La instrucción printf es una función de biblioteca que imprime la salida del terminal.

Para imprimir una secuencia de caracteres deben ser encerradas entre comillas. Por el momento solo usaremos el de carácter string. La secuencia \n en el string, es en C notación para indicar nueva línea (i.e.) después de escribir, el cursor avanza al margen izquierdo de la próxima línea; un printf nunca suministra una nueva línea automáticamente.

El ejemplo anterior también puede ser escrito como :

```
main ()
{
    printf ("hola, ");
    printf ("Freddy");
    printf ("\n");
}
```

para producir una salida idéntica.

Note que \n representa un solo carácter. Entre los varios otros escapes de secuencia, que C posee, están \t para TAB, \b para espacios en blanco, \" para doble cremilla y \\ para el backslash.

1.2 ARITMETICA Y VARIABLES

El próximo programa imprime una tabla de temperatura Fahrenheit y su equivalencia a Celsius usando la formula :

```
C = ( 5/9 )*( F-32 )
```

```
main ()
{
    int lower,upper,step;
    float fahr,celsius;
    lower = 0;
    upper = 300;
    step = 20;
    fahr = lower;
    while ( fahr <= upper )   {
        celsius = (5.0 / 9.0)*(fahr - 32.0);
        printf ("%4.0f %6.1f\n",fahr,celsius);
        fahr = fahr + step ;
    }
}
```

En C todas las variables deben ser declaradas antes de usar, para el comienzo de la función, antes de una declaración ejecutable.

Una declaración consiste de una expresión y una lista de variables, como en :

```
int lower , upper, step;
float fahr, celsius;
```

La declaración int indica que las variables son enteras; float representada por punto flotante, es decir nmeros que pueden tener parte fraccionaria. La precisión de ambas depende de la maquina que este usando. C entrega varias otras expresiones básicas además de int y float :

char	carácter - un solo byte
short	entero corto
long	entero largo
double	doble precisión de punto flotante

En el programa de calculo y conversión de temperatura empieza con los asignamientos :

```
lower = 0;
upper = 300;
step = 20;
fahr = lower;
```

De esta manera se fijan las variables para sus valores iniciales. Cada declaración individual es terminada en punto y

coma.

Cada línea de la tabla es calculada de la misma manera, por tanto se usa un loop que se repite una vez por línea; este es el propósito de la declaración while que es la misma que se usa en otros lenguajes.

Recomendamos escribir solo una declaración por línea, y usualmente permitiendo blancos alrededor de los operadores; la posición de las llaves es muy importante.

La temperatura en Celsius es calculada Y asignada a la variable celsius por la declaración :

```
celsius = ( 5.0 / 9.0 ) * ( fahr - 32.0 );
```

La razón para usar 5.0 / 9.0 en lugar de la simple forma 5/9 es que en C como en muchos otros lenguajes, la división de enteros es truncada.
Un punto decimal en una constante indica que esta es de punto flotante, así es como 5.0 / 9.0 es 0,555.... que es lo que se desea. También escribimos 32.0 en vez de 32, aunque sin embargo fahr es una variable float, 32 seria automáticamente convertido a float antes de la sustracción.

Veamos ahora el asignamiento

```
fahr = lower;
```

y la pregunta

```
while  (fahr <= upper)
```

Ambas palabras como es de esperar el int es convertido a float antes que la operación es hecha.

La siguiente declaración muestra un trozo de como imprimir variables :

```
printf("%4.0f %6.1f\n"fahr,celsius);
```

La especificación %4.0f quiere decir que un numero de punto flotante será imprimido en un espacio de ancho al menos de cuatro caracteres, con ningn dígito después del punto decimal,
%6.1f describe otro numero que al menos ocupara 6 espacios para su parte
entera, con un dígito después del punto decimal,analogo a los F6.1 de Fortran o el F(6.1) de PL/1.

Parte de una especificación puede ser omitida : %6f quiere decir que el numero es escrito al menos en 6 caracteres de ancho; %.2f quiere decir 2 espacios después del punto decimal. La anchura no es forzada si escribimos meramente %f ya que esto quiere decir que el numero a imprimir es de punto flotante. printf también reconoce a %d para enteros, %x para hexadecimal, %c para caracteres, %s para string, y %%
para %.

En toda construcción de printf, % es el primer argumento y todos los demás deben ponerse en fila, propiamente por numero y expresión o Ud. obtendrá respuestas sin sentido.

Cabe hacer notar que printf no es una instrucción propiamente del lenguaje C sino una función til que es parte de las bibliotecas standares o subrutinas que son normalmente accesibles en programas C.

1.3 LA DECLARACION FOR

Como Ud. podrá esperar, hay abundantes y diferentes maneras de escribir un programa; veamos una diferencia en el programa anterior :

```
main()
{
  int fahr;
  for (fahr = 0; fahr <= 300; fahr = fahr + 20)
  printf ("%4d %6.1f\n",fahr,(5.0 / 9.0)*(fahr - 32));
}
```

Esto produce la misma respuesta, pero es ciertamente diferente. El mayor cambio es la eliminación de la mayoría de las variables; solo fahr permanece como un int. Los limites inferior, superior y el incremento aparecen solo como constantes en la declaración for. Así mismo se tiene una nueva construcción, y la expresión que calcula la temperatura en celsius aparece como un tercer argumento del printf en el
lugar de ser una declaración separada.

Este ultimo cambio es un ejemplo de una regla completamente general
en C, en algn contexto esto es permisible para el uso de valores de alguna variable o de alguna expresión, se puede usar una expresión de cualquier tipo, ya que el tercer argumento es punto flotante para la pareja del %6.1f.

El for es una generalización del while; nótese que las tres partes constantes están separadas por punto y coma.

La forma del loop puede ser una sola declaración, o un grupo de declaraciones encerradas en paréntesis de llave.

1.4 CONSTANTES SIMBOLICAS

Afortunadamente C nos provee una manera para evitar nmeros mágicos, tales como 300 y 20. Con la construcción #define, en el comienzo de un programa puede definir un numero o constante simbólica.
Después de esto el compilador sustituirá todo sin cotizar la ocurrencia de los nombres. Esto no es limitado para los nmeros.

```
#define LOWER 0     /* limite menor */
#define UPPER 300   /* limite superior */
#define STEP 20     /* incremento */


main()
{
    int fahr;

for (fahr = LOWER;fahr <= UPPER;fahr = fahr + STEP)
printf ("%4d %6.1f\n",fahr,(5.0/9.0)*(fahr-32));

}
```

Las cantidades LOWER, UPPER y STEP son constantes, por lo tanto ellas no aparecen en declaraciones.

1.5 UNA COLECCION DE PROGRAMAS UTILES

Vamos ahora a considerar una serie de programas relacionados para hacer operaciones simples sobre datos de caracteres de I/O.

La biblioteca standard esta provista de funciones para leer y escribir un carácter al mismo tiempo, getchar() va a buscar la próxima entrada del carácter al mismo tiempo que esta es llamada:

```
c = getchar();
```

La variable c contiene el próximo carácter de entrada. Los caracteres normalmente vienen del terminal, pero no veremos esto hasta el cap.7.

La función putchar(c) es el complemento de getchar :

```
putchar(c);
```

imprime el contenido de la variable c en algn medio de salida, usualmente en el terminal. Llamar al putchar y printf puede ser intercalado; la salida aparecerá en el orden que las llamadas son hechas.

Como con printf, no hay nada acerca del getchar y putchar. Ellas no son parte del lenguaje C, pero son universalmente aprovechables.

COPIANDO ARCHIVOS

Dados getchar y putchar, Ud. puede escribir una sorprendente cantidad de códigos tiles sin conocer algo mas acerca de I/O. El simple ejemplo es un programa que copia su entrada a su salida un carácter a la vez.

Aquí esta el programa :

```
main()  /* copia entrada a la salida, 1ra versión */

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

El operador relacional != significa "distinto".

El problema principal es detectar el fin de entrada. Por convención, getchar devuelve un valor que no es un carácter valido cuando lo encuentra al final de la entrada; en esta forma los programas pueden detectar cuando ellos agotan la entrada. La nica complicación, un serio fastidio, es que hay dos convenciones en uso comn acerca de que este es realmente el valor del fin de archivo. Hemos diferido la emisión para usar el nombre simbólico EOF para el valor, cualquiera que
pueda ser. En la practica, EOF será -1 o 0 así el programa debe ser precedido por uno de los apropiados

```
#define EOF -1
```

o

```
#define EOF  0
```

Usando la constante simbólica EOF para representar el valor que getchar devuelve cuando el fin de archivo ocurre, estamos seguros que solo una cosa en el programa depende en el especifico valor numérico.

También decoramos c para ser un int, no un char, así puede esperar el valor que getchar devuelve. Así como veremos en el **CAPITULO** 2, este
valor es actualmente un int, ya que debe ser capaz de representar EOF en
adición a todos los posibles char's.

En C, algunos asignamientos tales como

```
c = getchar()
```

puede ser usado en alguna expresión; su valor es simplemente el valor que este siendo asignado a la izquierda. Si el asignamiento de un carácter para c es puesto dentro de la parte de la pregunta de un while el programa que copia archivos puede ser escrito como

```
main ()     /* copia entrada a la salida; segunda
               versión  */
{
    int c;

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

El programa entrega un carácter, asignándolo a c, y luego pregunta si el carácter ha encontrado el fin de archivo. El while termina cuando se ha detectado.

Esta versión centraliza el input - hay ahora una sola llamada para el getchar - y disminuye el programa. Anidando un asignamiento en una
pregunta es uno de los lugares donde C permite una preciosa brevedad.

Esto es importante para reconocer que el paréntesis alrededor del asignamiento dentro de la condición es realmente necesario. La presencia de != es mayor que la de =, lo cual significa que en la ausencia de paréntesis la pregunta != será hecha antes del asignamiento =. Así la instrucción

```
c = getchar() != EOF
```

es equivalente a

```
c = (getchar() != EOF)
```

Esto tiene el efecto indeseado de poner c en 0 o 1, dependiendo en si la llamada de getchar encuentra el fin de archivo o no. (Mas de esto en cap.2.)

CONTANDO CARACTERES

El próximo programa cuenta caracteres; esta es una elaboración mas fina del programa anterior.

```
main ()   /* cuenta caracteres en la entrada */
{
    long nc;
    nc = 0;

    while (getchar() != EOF)
        ++nc;
    printf (" %1d\n",nc);
}
```

La instrucción

```
++nc;
```

muestra un nuevo operador, ++, que significa incremento en uno. Ud. puede escribir nc=nc+1 pero ++nc es mas conciso y frecuentemente mas
eficiente. Hay una correspondencia con el operador -- que decrementa en uno. Los operadores ++ y -- pueden ser un operador prefijo (++nc) o
sufijo (nc++); estas dos formas tienen diferente valor en las expresiones como mostraremos en el cap.2, ambos incrementan nc. Por
el momento nos quedaremos con el prefijo.

El programa que cuenta caracteres acumula su cuenta en una variable long en vez de un int. Para hacer frente, incluso a nmeros grandes, Ud. puede usar double (doble largo de float). Usaremos la instrucción for en vez de un while para ilustrar una manera alternativa de escribir el loop.

```
main ()  /* cuenta caracteres en la entrada */
{
    double nc;


    for (nc = 0; getchar () != EOF ; ++nc)
        ;
    printf ("%.0f\n",nc);
}
```

El printf usa %f para ambos float y double; %.0f suprime la parte fraccionaria.

El cuerpo del ciclo for esta vacío, porque todo el trabajo es hecho en la pregunta y la reinicialización. Pero las reglas gramaticales de C requiere que un for tenga un cuerpo. El punto y coma aislado es una declaración técnicamente nula, esta aquí para satisfacer el requerimiento. La ponemos en línea separada para hacerla mas visible.

Antes de dejar el programa que cuenta caracteres, obsérvese que si la entrada no contiene caracteres, el while o for abandona la pregunta en la primera llamada a getchar, y así el programa produce cero. Esta es una observación importante. Una de las cosas agradables acerca del while y for es que ellos preguntan hasta el tope del loop, antes del procedimiento con el cuerpo. Si no hay nada que hacer, nada es hecho, aun si esto quiere decir que nunca vaya através del cuerpo del loop. El while y for ayudan a asegurar que ellos hacen cosas razonables con un
limite de condiciones.

CONTANDO LINEAS

El próximo programa cuenta líneas en la entrada. La entrada de líneas asume que son terminadas por el carácter newline \n.

```
main ()  /* cuenta líneas en la entrada */
{
    int c ,nl;
    nm = 0;

    while (( c = getchar()) != EOF) {
        if (c == '\n')
            ++nl;

    printf ("%d\n ",nl);
 }
```

El cuerpo del while ahora consiste de un if, el cual controla la vuelta y el incremento de ++nl. La instrucción if, pregunta por la condición entre paréntesis, y si es verdadera, hace lo que la instrucción siguiente diga (o un grupo de declaraciones entre llaves).

El doble signo == es notación en C para "es igual a" (como en Fortran .EQ.). Este símbolo es usado para distinguir la igualdad de las preguntas con el signo = usado para asignamiento.

Cualquier unidad de carácter puede ser escrito entre dos comillas, este es el llamado carácter constante.

La secuencia de escape usada en string de caracteres son también legales en caracteres constantes, así en preguntas y expresiones aritméticas, '\n' representa el valor del carácter newline. Ud. debería notar cuidadosamente que '\n' es un simple carácter, y en expresiones es equivalente a un simple entero; de otra forma, "\n" es un carácter de string que contiene por casualidad solo un carácter. El tópico de string versus caracteres es discutido mas adelante en el **CAPITULO** 2.

CONTANDO PALABRAS

El cuarto de nuestra serie de programas tiles cuenta líneas, palabras y caracteres, con la definición que una palabra es alguna secuencia de caracteres que no contiene un blanco, TAB o newline. (Esta es una versión del utilitario wc de UNIX.)

```
#define YES    1
#define NO     0

main()   /* cuenta líneas, palabras, caracteres en la
                    entrada */
{
    int c, nl, nw, nc, inword;
    inword = NO;
    nl = nw = nc = 0;

    while ((c = getchar()) != EOF)     {
        ++nc;
    if(c == '\n')
        ++nl;
    if (c == ' ' || c == '\n' || c == '\t')
        inword = NO;
        else if (inword == NO)   {
            inword = YES;
            ++nw;
        }
    }
    printf (" %d %d %d\n", nl, nw, nc);
}
```

Cada vez que el programa encuentra el primer carácter de una palabra, este es contado. La variable inword registra si el programa esta actualmente en una palabra o no, inicialmente este no esta en una palabra, al cual es asignado el valor NO. Preferimos los símbolos constantes, YES y NO para los valores literales uno y cero porque hacen que el programa sea mas legible. Ud. también encontrara que es fácil hacer cambios extensivos en programas donde los nmeros aparecen solo como constantes simbólicas.

La línea

```
nl = nw = nc = 0;
```

pone en cero a las tres variables. Esto no es un caso especial, pero una consecuencia de la realidad que un asignamiento tiene un valor y un asignamiento asociado de derecha a izquierda. Es como si hubiésemos escrito :

```
nc = (nl = (nw = 0));
```

El operador || significa un OR, así la línea :

```
if (c == ' ' || c == '\n' || c == '\t')
```

quiere decir "si c es un blanco o c es un newline o c es un TAB" (la secuencia de escape \t es una representación visible del carácter TAB). Hay también una correspondencia con el operador && para el AND. Al conectar expresiones por && o || son evaluadas de izquierda a derecha, y
esto garantiza que la evaluación no pasara tan pronto como la verdad o falsedad es comprendida. Así si c contiene un blanco, no hay necesidad de preguntar si contiene un newline o TAB, así estas preguntas no son hechas. Esto no es particularmente importante aquí, pero es mas significativo en muchas situaciones complicadas, como veremos pronto.

El ejemplo también muestra la declaración else de C, la cual especifica una alternativa de acción para ser hecha si la condición parte de un if siendo falsa. La forma general es :

```
if ( expresión )
    declaracion-1
else
   declaracion-2
```

Una y solo una de las declaraciones asociadas con un if-else es hecha. Cada una de las declaraciones puede ser completamente complicada.

1.6 ARREGLOS

Escribiremos un programa para contar el numero de ocurrencias de cada dígito, espacios en blanco, y otros caracteres. Esto es artificial, pero nos permite usar varios aspectos de C en un programa.

Hay doce categorías de input, es conveniente usar un arreglo para tener el numero de ocurrencias de cada dígito, en vez de usar diez variables individuales. Aquí esta una versión del programa :

```
main ()
{
    int c,i,nwhite,nother;
    int ndigit[10];
    nwhite = nother = 0;
    for (i = 0;i < 10;++i)
        ndigit[i] = 0;
        while ((c = getchar()) != EOF)
        if (c >= '0' && c <= '9')
            ++ndigit[c-'0'];
        else if (c == ' ' || c == '\n' || c == '\t')
                ++nwhite;
            else
                ++nother;
        printf ("dígitos = ");
        for ( i = 0; i < 10; ++i)
            printf ("%d ",ndigit[i]);
    printf ("\n espacios en blanco = %d,otros = %d\n",
            nwhite,nother);
}
```

La declaración int ndigit[10]; declara que el arreglo es de enteros. El arreglo siempre es subindicado en el comienzo por cero en C; por lo tanto los elementos son ndigit[0],..., ndigit[9].

Una subindicacion puede ser alguna expresión entera, que incluye variables enteras como i, y constantes enteras.

Este programa hace un relieve importante en las propiedades de los caracteres en la representación de los dígitos. Por ejemplo, la pregunta

```
if (c >= '0' && c <= '9')
```

determina si el carácter en c es un dígito. Si este lo es, el valor numérico de ese dígito es :

```
c - '0'
```

Esto trabaja solo si '0', '1', etc. son positivos en orden creciente si hay solo dígitos entre '0' y '9'. Afortunadamente esto es verdadero para todos los conjuntos de caracteres convencionales.

Por definición, la aritmética envuelve a char's e int's convirtiendo todo en int antes de proceder, Así las variables char y constantes son escencialmente idénticas a int's en el contexto aritmético. Esto es bastante natural y conveniente; por ejemplo c - '0' es un entero con valor entre cero y nueve correspondiendo a los caracteres '0' y '9' almacenado en c, es así una subindicacion valida para el arreglo ndigit.

La decisión referente a si un carácter es un dígito, un espacio en blanco, o algo mas es hecho por la secuencia

```
if ( c >= '0' && c <= '9' )
    ++ndigit[c-'0'];
else if (c == ' ' || c == '\n' || c == '\t')
    ++nwhite;
else
    ++nother;
```

El modelo

```
if (condición)
    instrucción
else if ( condición)
    instrucción
else
    instrucción
```

ocurre frecuentemente en programas como una forma de expresar una multidecision. El código es simplemente leído desde el tope hasta que alguna "condición" es satisfecha; hasta ese punto la correspondiente "instrucción" es ejecutada, y la construcción entera es finalizada. (La "instrucción" en curso pueden ser varias instrucciones encerradas en paréntesis.) Si ninguna de las condiciones es satisfecha, la "instrucción" después del else final es ejecutada si esta presente. Si el else final y "instrucción" son omitidos (como en el programa que cuenta palabras), la acción no toma lugar. Allí puede estar un numero arbitrario de

```
else if (condición)
    instrucción
```

grupos entre el if inicial y el else final. Como un estilo de modelo, es aconsejable formatear esta construcción como hemos mostrado.

1.7 FUNCIONES

En C una función es equivalente a una subrutina o función en Fortran, o una procedure en PL/1, Pascal, etc. Una función proporciona una manera conveniente para envasar algn calculo en una caja negra. Las funciones son realmente la nica manera de hacer frente a la potencial complejidad de grandes programas. Con funciones construidas
en forma apropiada es posible ignorar "como" es hecha una tarea; sabiendo que esta hecho es suficiente. C esta hecho para servirse de funciones fáciles, convenientes y eficientes; Ud. vera a menudo llamar una función solo una vez en unas pocas líneas, justo porque esto clarifica alguna parte del código.

Hasta ahora hemos usado funciones como printf, getchar y putchar que han sido proporionadas por nosotros; es tiempo de esribir unos pocos
de nuestra propiedad. Ya que C no tiene operador de exponenciacion parecido al ** de Fortran o PL/1, ilustraremos el mecanismo de esta función escribiendo la definición de una función power(m,n) que eleva un
entero m a un entero positivo n. Así el valor power (2,5) es 32. Esta función ciertamente no hace todo el trabajo que hace ** ya que es manejable solo con powers positivos (n > 0) de enteros menores que 30, pero es mejor confundir un problema a la vez.

Aquí esta la función power y un programa principal para ejercitarlo, así Ud. puede ver toda la estructura a la vez.

```
main ()
{
    int i;
    for (i = 0; i < 10; ++i)
  printf ("%d %d %d\n",i,power(2,i),power(-3,i));
}

power (x,n)
int x, n;
{
    int i,p;
    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * x;
    return (p);
}
```

Cada función tiene la misma forma :

```
nombre (lista de argumentos, si hay algunos)
declaraciones de argumentos, si hay algunas
{
    declaraciones
    instrucciones
}
```

Las funciones pueden aparecer en uno u otro orden, y en el archivo original o en dos. En curso, si el original aparece en dos archivos, Ud. tendrá que decirle mas al compilador y cargarlo si este aparece en uno, pero ese es un modelo del sistema operativo, no un atributo del lenguaje. Por el momento asumiremos que ambas funciones están en el mismo archivo, así cualquier cosa que Ud. haya aprendido acerca de correr programas en C no cambiara.

La función power es llamada dos veces en la línea :

```
printf ("%d %d %d\n",i,power(2,i),power(-3,i));
```

Cada llamada da paso a dos argumentos para power, los cuales cada vez devuelven un entero para ser formateado e impreso. En una expresión, power(2,i) es un entero justamente como son 2 e i.(No todas las funciones producen un valor entero; discutiremos esto en el Cap. 4)

En power los argumentos han de ser declarados apropiadamente, así sus tipos seran comprendidos. Esto es hecho por la línea

```
int x, n;
```

que sigue al nombre de la función. La declaración de argumentos va entre la lista de argumentos y la llave abierta en la izquierda; cada declaración es terminada por un punto y coma. Los nombres usados por
power para sus argumentos son puramente "locales" a power, y no accesible a alguna otra función: otras rutinas pueden usar el mismo nombre sin dificultad. Esto es cierto para las variables i y p : la i en power no esta relacionada con la i en main.

El valor que power calcula es devuelto a main por la instrucción return, lo que es justamente como en PL/1. Alguna función puede ocurrir dentro del paréntesis. Una función no necesita devolver un valor; una instrucción return no causa una expresión de control, pero no utiliza un valor, para ser devuelto por la llamada, así desciende al final de la función alcanzando el termino en la llave derecha.

1.8 ARGUMENTOS - LLAMADA POR VALORES

Un aspecto de las funciones en C pueden ser desconocidas para programadores que han usado otros lenguajes, particularmente Fortran y PL/1. En C, todo argumento de función es traspasado "por su valor". Esto significa que la función llamada tiene dados los valores de sus argumentos en variables temporales (actualmente en un stack) mas bien que sus direcciones. Esto conduce a algunas diferentes propiedades que son vistas con "llamada por referencia" con lenguajes como Fortran y PL/1, en que la rutina llamada es guiada a la direccion del argumento, no su valor.

La distincion principal es que en C, la función llamada no puede alterar a una variable en la función llamada; esto puede alterar su privacidad, es una copia temporal.

La llamada por valores, usualmente esta dirigida a programas mas compactos con menos variables externas, porque los argumentos pueden ser
tratados como variables locales, convenientemente inicializadas en la rutina llamada.

Por ejemplo, aquí esta una versión de power que hace uso de esta realidad.

```
power(x,n)  /* eleva x a la enesima potencia; n>0,
               segunda versión */
int x,n;
{
    int p;

    for (p = 1; n > 0; --n)
        p = p * x;
    return(p);
}
```

El argumento n es usado como una variable temporal, y es decrementado hasta que llega a ser cero; no hay un largo necesario para la variable i. Todo lo que es hecho para n dentro de power no tiene efecto en el argumento con que power fue originalmente llamado.

Cuando es necesario, es posible arreglar una función para modificar una variable en una rutina de llamada. El que llama debe proporcionar la "direccion" de la variable a ser colocada (técnicamente un "puntero" para la variable), y la función llamada debe declarar el argumento que será un puntero y referencie indirectamente a traves de la variable actual. Cubriremos esto en detalle en el cap.5.

Cuando el nombre de un arreglo es usado como un argumento, el valor
pasado a la función es actualmente la localizacion o direccion del comienzo del arreglo.(No hay copia de elementos del arreglo) Suscribiendonos a este valor, la función puede accesar y alterar algn elemento del arreglo. Este es el tópico de la próxima seccion.

1.9 ARREGLOS DE CARACTERES

Probablemente el tipo de arreglo mas comn en C es el de caracteres. Para ilustrar el uso de arreglo de caracteres, y funciones que manipularemos, escribamos un programa que lee un conjunto de líneas
e imprime la mas larga. La estructura del programa es bastante simple :

```
while ( hay otra línea?)
    if ( es mas largo que el anterior?)
        grabarla y su largo
imprima largo y línea
```

Esta estructura divide el programa en partes. Una parte entrega una nueva línea, otra pregunta, otra graba, y el resto controla el proceso.

De acuerdo con lo anterior escribiremos una función separada, getline para ir a buscar la próxima línea de entrada; esta es una generalización de getchar. Para que la función sea til en otros contextos, probaremos tratando de hacerlo tan flexible como sea posible.

En lo minimo, getline tendrá que devolver una indicacion acerca del posible EOF; una utilidad mas general seria la construcción para devolver el largo de la línea o el cero si el EOF es encontrado. Cero nunca es una longitud valida para una línea, ya que toda línea tiene al menos un carácter; aun una línea conteniendo solo un newline tiene largo
1.

Cuando encontramos una línea que es mas larga que las anteriores, esto debe ser guardado en alguna parte. Esto sugiere una segunda función, copy, para copiar la nueva línea en un lugar seguro.

Finalmente, necesitaremos un programa principal para el control de getline y copy. Aquí esta el resultado.

```
#define MAXLINE 1000 /* largo maximo de la línea
                        de entrada */
main()   /* encuentra la línea mas larga */
```

```
{
    int len;  /* largo de la línea en curso */
    int max;  /* largo maximo visto hasta ahora */
    char line  [MAXLINE]; /* línea de entrada actual
                             */
    char save  [MAXLINE];  /* línea mas larga, grabada
                             */

    max = 0;
    while ((len = getline(line, MAXLINE)) >0)
        if (len > max)      {
            max = len ;
            copy (line,save);
        }
    if (max > 0)   /* hubo una línea */
        printf ("%s",save);
}




getline(s,lim) /* entrega la línea en s, devuelve el
                          largo */
char [s];
int lim;
{
    int c,i;
    for (i=0;i<lim-1 && (c=getchar()) != EOF &&
            c!='\n';++i)
        s[i] = c;
    if (c == '\n')  {
        s[i] = c;
        ++i;
    }
    s[i] ='\0';
    return(i);
}

copy(s1,s2)   /* copia s1 en s2; asume s2 bastante
                    grande */
char s1[], s2[];
{
    int i;
    i=0;

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

main y getline se comunican a traves de un par de argumentos y un valor devuelto. En getline, los argumentos son declarados por las líneas

```
    char s  ;
    int lim;
```

las cuales especifican que el primer argumento es un arreglo, y el segundo es un entero. La longitud del arreglo s no esta especificada en getline ya que esta determinada en main. getline usa return para enviar un valor de vuelta al llamado, justo como lo hizo la función power. Algunas funciones devuelven un valor til; otras como copy son solo usadas para sus efectos y no devuelven un valor.

getline pone el carácter \O (el carácter nulo cuyo valor es cero) para el fin del arreglo, para marcar el fin del string de caracteres. Esta convención es también usada por el compilador C : cuando un string constante parecido a

```
"hello\n"
```

es escrito en un programa C, el compilador crea un arreglo de caracteres conteniendo los caracteres del string, y lo termina con \0 así esas funciones tales como printf pueden detectar el final.

El formato de especificación %s en printf espera un string representado en esta forma. Si Ud. examina copy, descubrira que esto también cuenta en realidad con que su argumento de entrada s1 esta terminado por \0, y esto copia los caracteres en el argumento de salida s2.(Todo esto implica que \0 no es parte de un texto normal)

Mencionamos de pasada que un programa tan chico como este presenta algunos problemas de construcción. Por ejemplo que debería hacer main si al encontrar una línea la cual es mas grande que su limite? getline trabaja oportunamente, es decir detiene la recoleccion de caracteres cuando el arreglo esta lleno, aun si el newline no ha sido visto. Preguntando por la longitud y el ultimo carácter devuelto, main puede determinar cuando la línea fue demasiado larga.

No hay manera de usar un getline para conocer el avance de cuan larga debe ser una línea de entrada, así getline es un obstaculo para overflow.

1.10 VARIABLES EXTERNAS

Las variables en main (line, save, etc.) son privadas o locales a main; porque ellas son declaradas dentro del main, otras funciones no pueden tener acceso directo a ellas. Lo mismo ocurre con las variables en otras funciones; por ejemplo, la variable i en getline no esta relacionada con la i en copy. Cada una de las variables locales va dentro de una rutina solo cuando la función en existencia es llamada y desaparecera cuando la función es dejada de lado. Es por esta razón que tales variables son usualmente conocidas como variables automaticas, siguiendo la terminologia de otros lenguajes usaremos el termino automatico de aquí en adelante para referirnos a estas variables locales dinamicas.(En el cap.4 discutiremos las clases de almacenaje estatico, en las cuales las variables locales retienen sus valores entre la invocacion de funciones)

Porque las variables automaticas van y vienen con la invocacion de una función, ellas no retienen sus valores desde una llamada a la próxima, y deben estar explicitamente sobre cada entrada de un set. Si ellas no están en un set, contendran basura.

Como una alternativa para variables automaticas, es posible definir variables, que son externas para todas las funciones, esto es, variables globales que pueden ser accesadas por el nombre de alguna función.(Este mecanismo es semejante al Fortran common o al pl/1 external) Porque las variables externas son globalmente accesibles, pueden ser usadas en vez de listas de argumentos para comunicar datos entre funciones. Además porque las variables permanecen en existencia permanentemente, mas bien que apareciendo y desapareciendo mientras que las funciones son llamadas y dejadas de lado, ellas retienen sus valores aun después que el set de funciones es ejecutado.

Una variable externa es definida fuera de una función. Las variables deben ser declaradas también en cada función que se desee accesar; esto puede ser hecho por una declaración explicita extern o implicita en el contexto. Para hacer mas concreta la discusion, nos permitiremos reescribir el programa de las líneas mas largas con line, save y max como variables externas. Esto requiere cambiar los llamados, declaraciones, y cuerpos de las tres funciones.

```
#define maxline 1000 /*largo maximo de entrada  */

char line [MAXLINE];  /* línea de entrada */
char save [MAXLINE];   /* línea mas larga grabada aquí
```

```
                                */
    int max ;               /* largo de la línea mas larga
                            vista hasta aquí */

    main ()     /* encuentra línea mas larga; versión
{                           especializada */

        int len;
        extern int max;
        extern char save[];

        max = 0;
        while ((len = getline()) > 0)
            if (len > max)  {
                max = len;
                copy();
        }
        if (max > 0)   /* aquí hubo una línea */
            printf ("%s",save);
    }


    getline ()     /* versión especializada */
    {

        int c,i;
        extern char line  ;

        for (i=0; i < MAXLINE-1
             && (c=getchar()) != EOF && c!='\n';++i)

            line [i] = c;
        if ( c == '\n')     {
            line [i] = c;
            ++i;
        }

        line [i] = '\0';
        return(i);
    }

    copy()   /* versión especializada  */
    {

        int i;
        extern char line[], save[];

        i = 0;
        while ((save[i] = line[i]) != '\0')
            ++i;
    }
```

Las variables externas en main, getline y copy son definidas por las primeras líneas del ejemplo anterior. Sintacticamente, las definiciones externas son justamente semejantes a las declaraciones que hemos usado previamente, pero desde que ellas ocurren fuera de las funciones, las variables son externas. Antes una función puede usar una variable externa, el nombre de la variable debe ser conocida para la función. Una manera de hacer esto es escribir una declaración extern en la función; la declaración es la misma de antes, excepto por el agregado de extern.

En ciertas circunstancias, la declaración extern puede ser omitida: si la definición externa de una variable ocurre en el archivo fuente antes de usarla en una función particular, entonces no hay necesidad de una declaración extern en la función. Las declaraciones extern en main, getline y copy son así redundantes. En efecto, en la practica comn es ubicar definiciones de todas las variables externas al comienzo del archivo fuente, y entonces omitir todas las declaraciones extern.

Ud. debería notar que hemos usado cuidadosamente las palabras declaración y definición cuando nos referimos a variables externas en esta seccion. Definición se refiere al lugar donde la variable es actualmente creada o asignada a un almacen, declaración se refiere al lugar donde la naturaleza de la variable es declarada pero no almacenada.

De esta forma, hay una tendencia para hacer todo con miras a una variable extern porque esto parece simplificar las comunicaciones (las listas de argumentos son cortas y las variables están siempre ahi cuando Ud. las requiera. Pero las variables externas están siempre ahi aun cuando Ud. no las requiera).

CAPITULO 2 TIPOS DE OPERADORES Y EXPRESIONES

Las variables y constantes son los datos objetos bsicos manipulados en un programa. Las declaraciones listan las variables a usar, y declaran de que tipo seran ellas y quizas cual es su valor inicial. Los operadores especifican que es hecho por ellas. Las expresiones combinan variables y constantes para producir nuevos valores.

2.1 NOMBRES DE VARIABLES

Hay algunas restricciones en los nombres de variables y constantes simbolicas. Los nombres son formados con letras y d¡gitos; el primer carácter debe ser una letra. El underscore (_) se cuenta como una letra, esto es til para mejorar la calidad legible de largos nombres.

Solo los primeros ocho caracteres de un nombre interno son significativos, aunque pueden utilizarse mas de ocho. Para nombres externos tales como nombres de funciones y variables externas, el n£mero
puede ser menor que ocho, porque los nombres externos son usados por diferentes assemblers y loaders. El apendice A lista los detalles. Además, las keywords como if, else, int, float, etc.,son reservadas, es decir, Ud. no las puede usar como nombre de variables. (A menos que se
use el underscore entre las letras)

Naturalmente es prudente elegir nombres de variables que signifiquen algo, que son relativas al propósito de la variable, y que son improbable de obtener una mezcla tipografica.

2.2 TIPOS DE DATOS Y LARGOS

Hay solo unos pocos tipos de datos bsicos en C :

char Un solo byte, capaz de retener un carácter en el set de caracteres local.
int Un entero, reflejando t¡picamente el largo natural de enteros en la mquina anfitriona.
float Punto flotante, simple precisi¢n.
double Punto flotante de doble precisi¢n.

En suma, hay un n£mero de cualidades que pueden ser aplicadas a enteros : short, long y unsigned. Los n£meros short y long se refieren a diferentes largos de enteros. Los n£meros unsigned obedecen a la ley de aritmética modulo '2 elevado a n', donde n es el n£mero de bits en un int; los n£meros unsigned son siempre positivos. Las declaraciones para estos calificativos se asemejan a:

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

La palabra "int" puede ser omitida en tales situaciones.
La precisi¢n de estas depende de la mquina en que sean manipuladas; la tabla de mas abajo muestra algunos valores representativos.

	DEC PDP-11 ASCII	HONEYWELL 6000 ASCII	IBM 370 EBCDIC	INTERDATA 8/32 ASCII
char	8 bits	9 bits	8 bits	8 bits
int	16	36	32	32
short	16	36	16	16
long	32	36	32	32
float	32	36	32	32
double	64	72	64	64

El intento es que short y long proporcionarian diferentes largos de enteros; int normalmente reflejara el largo mas natural para una mquina en particular. Como Ud.puede ver, cada compilador es libre para interpretar short y long como le sea apropiado a su hardware. Sobre todo Ud. debería contar con que short no es mas largo que long.

2.3 CONSTANTES

int y float son constantes que ya han sido dispuestas, excepto para notar que la usual

```
123.456e-7
```

o

```
0.12E3
```

notaci¢n cientifica para float es también legal. Cada constante de punto flotante es tomada para ser double, así la notaci¢n "e" sirve para ambos.

Las constantes largas son escritas en el estilo 1326. Un entero constante comn que es demasiado grande para encajar en un int es también tomado para ser un long.

Hay una notaci¢n para constantes "octal y hexadecimal"; un cero delante de una constante int implica un octal, un 0x o 0X delante indica que es un hexadecimal. Por ejemplo, el decimal 31 puede ser escrito como 037 en octal y 0x1f o 0X1F en hexadecimal. Las constantes hexadecimal y octales pueden estar seguidas por L, para hacerlas long.

Un carácter constante es un carácter individual escrito entre comillas simples, como en 'x'. El valor de un carácter constante es el valor num,rico del set de caracteres de mquina. Por ejemplo, en el set de caracteres ASCII el carácter cero, o '0', es 48, y en EBCDIC '0' es 240, ambas son muy diferentes al valor num,rico de 0. Escribiendo '0' en lugar de un valor num,rico semejante a 48 o 240 hace al programa independiente del valor en particular.

Los caracteres constantes participan en operaciones num,ricas justo como cualquier otro n£mero, aunque ellos son mas frecuentemente usados
en comparaciones con otros caracteres. Mas tarde una secci¢n trata de las reglas de conversi¢n.

Los caracteres contantes participan en operaciones num,ricas tal como en algunos otros n£meros, aunque ellos son frecuentemente mas usados en comparaciones con tros caracteres.

Ciertos caracteres no-graficos pueden ser representados en caracteres constantes por secuencia de escape semejantes a \n (newline), \t (TAB), \0 (nulo), \\ (backslash), \' (comilla simple), etc., los cuales parecen dos caracteres, sin embargo son uno solo.

Una expresi¢n constante es una expresi¢n que involucra solamente constantes. Tales expresiones son evaluadas por el compilador, mas bien que al ser ejecutadas, y por consiguiente pueden ser usadas en algn lugar que puede ser constante, como en:

```
#define MAXLINE 1000
char line [MAXLINE+1];
```

o

```
seconds = 60 * 60 * hours;
```

Un string constante es una secuencia de cero o mas caracteres entre comillas, como en :

```
" Rolando alias el GALAN DE CIISA " <-- STRING
```

o

```
"" /* un string nulo */
```

Las comillas no son parte del string, pero solo sirven para delimitarlo. Las mismas secuencias de escape utilizadas por caracteres constantes son aplicables en strings; \" representa el carácter de doble comilla.

Técnicamente un string es un arreglo cuyos elementos son caracteres simples. El compilador automáticamente ubica el carácter nulo \0 al final de cada string, así los programas pueden encontrar convenientemente el final. Esta representaci¢n significa que no hay un limite real para cuan largo puede ser un string, pero hay programas para buscar y determinar completamente su largo.

La siguiente funci¢n strlen(s) devuelve el largo del string de caracteres s excluyendo el ultimo \0.

```
strlen(s) /* devuelve largo del string s */
char s[];
{
    int i;
    i=0;
    while(s[i] != '\0')
        ++i;
    return(i);
}
```

Debe uno ser cuidadoso para distinguir entre un carácter constante y un string constante de un solo carácter : 'x' no es lo mismo que "x". El primero es un carácter simple, usado para producir el valor num,rico de la letra x en el set de caracteres de mquina. El segundo es un string que contiene un solo carácter (la letra x) y un \0.

2.4 DECLARACIONES

Todas las variables deben ser declaradas antes de ser usadas, aunque ciertas declaraciones pueden ser hechas implicitamente en el contexto. Una declaraci¢n especifica un tipo y es seguida por una lista de una o mas variables de ese tipo, como en :

```
int lower, upper, step;
char c, line[1000];
```

Las variables pueden ser distribuidas entre varias declaraciones en cualquier forma; las listas de arriba son equivalentes a :

```
int lower;
int upper;
int step;
char c;
char line[1000];
```

Esta forma toma mas espacio, pero es conveniente para agregar un comentario a cada declaraci¢n o para modificaciones subsecuentes.

Las variables pueden también ser inicializadas en su declaraci¢n, aunque hay algunas restricciones. Si el nombre es seguido por un signo igual y una constante que sirve como inicializador, como en:

```
char backslash = '\\';
int i = 0;
float eps =  1.0e-5;
```

Si la variable en cuestion es externa o estatica, la inicializaciones hecha solamente una vez, conceptualmente antes de que el programa comience sus ejecuci¢n. Explicitamente la inicializaci¢n automatica de variables es inicializada al momento que la funci¢n es llamada. Las variables automaticas para las cuales no hay inicializador explicito, tienen basura. Las variables externas y estaticas son inicializadas en cero por omisi¢n, pero, de todos modos esto es un buen estilo para el estado de la inicializaci¢n.

2.5 OPERADORES ARITMETICOS

Los operadores aritmeticos binarios son +, -, +, /, y el operador de modulos %.

La divisi¢n de enteros trunca alguna parte fraccional. La expresi¢n

```
x % y
```

produce el resto cuando x es dividido por y, y así es cero cuando y divide exactamente a x . Por ejemplo, un "leap year" es divisible por 4 pero no por 100, excepto que "year" sea divisible por 400 son "leap years". Por lo tanto

```
if (year % 4 == 0 && year %  100 != 0 || year % 400 == 0)
    es un ano bisiesto
else
    no lo es
```

El operador % no puede ser aplicado a float o double.

Los operadores + y - tienen la misma precedencia, la que es menor que la precedencia de *, /, y %, que son mas bajos que el unario -. (Una tabla al final de este **CAPITULO** sumariza la precedencia y asociatividad para todos los operadores.)

El orden de evaluaci¢n no es especificado por asociatividad o conmutatividad de operadores como * y +; el compilador debe redistribuir un paréntesis que envuelve a uno de estos computos.

Así a+(b+c) puede ser evaluado como (a+b)+c. Esto raramente hace alguna diferencia, pero si un orden particular es requerido deben explicitarse las variables temporales que deban ser usadas.

2.6 OPERADORES LOGICOS Y DE RELACION

Los operadores de relaci¢n son

```
> >= < <=
```

Todos ellos tienen la misma precedencia. Les siguen en precedencia los
operadores de igualdad :

```
==  !=
```

los cuales tienen la misma precedencia. Estas relaciones tienen menor precedencia que los operadores aritmeticos, así, expresiones parecidas a i < lim-1 son tomadas como i < (lim-1), como debería ser esperado.

Mas interesante son los conectores logicos && y ||. Las expresiones conectadas por && o || son evaluadas de izquierda a derecha,
y la evaluaci¢n detenida tan pronto como la verdad o falsedad del resultado es conocida. Estas propiedades criticas para la estructura de los programas en donde trabajan. Por ejemplo, este es un loop de la funci¢n getline que escribimos en el cap.1.

```
for ( i=0; i<lim-1 && (c=getchar()) != '\n' &&
        c != EOF; ++i)
    s[i] = c;
```

Claramente, antes de leer un nuevo carácter es necesario chequear que hay un espacio para marcarlo en el arreglo s, así la pregunta i < lim-1 debe hacerse primero. No solo esto, pero si esta pregunta falla, no deberiamos ir a leer otro carácter.

Similarmente, seria desafortunado si c fuera evaluado con EOF antes de que getchar fuera llamada: la llamada debe ocurrir antes de que el carácter en c sea evaluado.

La precedencia de && es mayor que la de ||, y ambas son menores que
los operadores de relaci¢n e igualdad, así expresiones como :

```
i < lim-1 && ( c = getchar()) != '\n' && c != EOF
```

no necesitan paréntesis extra. Pero ya que la precedencia de != es mayor que un asignamiento los paréntesis son necesarios en

```
(c =getchar()) != '\n'
```

para conseguir el resultado deseado.

El operador de negaci¢n unario ! convierte un operando verdadero o no-cero, en un cero; y un operando falso o cero en 1. Se usa comunmente en construcciones como

```
if (!inword)
```

en vez de

```
if (inword == 0)
```

Es dificil generalizar acerca de cual forma es mejor. Construcciones mas complicadas que !inword pueden ser dificiles de entender.

2.7 TIPOS DE CONVERSION

Cuando operandos de diferentes tipos aparecen en expresiones, son convertidos a un tipo comn de acuerdo a un reducido n£mero de reglas. En general, solo las conversiones que suceden automáticamente son aquellas que tienen sentido, tal como convertir un entero a float en una expresi¢n como f + i. Las expresiones que no tienen sentido, como usar un float como un subindice, no son permitidas.

Primero, char's e int's pueden ser libremente entremezclados en expresiones aritméticas :cada char en una expresi¢n es automáticamente convertido a un int. Esto permite una considerable flexibilidad en cierta clase de caracteres. Una es ejemplificada por la funci¢n atoi, que convierte un string de d¡gitos a su equivalente num,rico.

```
atoi(s)    /* convierte s a entero */
char s[];
{

    int i,n;

    n = 0;
    for (i=0; s[i] >= '0' && s[i]  <= '9'; ++i)
        n = 10 * n + s[i] - '0';
    return(n);
}
```

Como ya vimos en el cap.1, la expresi¢n

```
s[i] - '0'
```

da el valor num,rico del carácter almacenado en s[i] porque los valores de '0', '1', etc., forman una secuencia de n£meros positivos contiguos en orden creciente.

Otro ejemplo de conversi¢n de char a int es la funci¢n lower, que convierte letras mayusculas a minusculas solo para caracteres ASCII, si el carácter no es una letra mayuscula, lower devuelve el mismo carácter ingresado.

```
lower(c)   /* convierte c a minuscula; solo ASCII  */
int c;
{
    if (c >= 'A' && c <= 'Z')
        return(c + 'a' - 'A');
    else
        return(c);
}
```

Hay un punto muy sutil acerca de la conversi¢n de caracteres a enteros.

El lenguaje no especifica si la variables de tipo char son cantidades indicadas o no indicadas. Cuando un char es convertido a un
int, puede esto aun producir un entero negativo? Desafortunadamente, esto varia de mquina a mquina, reflejando diferencias en la arquitectura. En algunas mquinas (PDP-11, por ejemplo), un char cuyo bit de mas a la izquierda es 1 será convertido a un entero negativo ("flag de extension"). En otras, un carácter es promovido a un int por suma de ceros al final de la izquierda, y así es siempre positivo.

La definici¢n de C garantiza que cualquier carácter en las mquinas standards el set de caracteres nunca será negativo; así estos caracteres pueden ser usados libremente en expresiones como cantidades positivas. Pero arbitrariamente modelos de bit almacenados en caracteres variables pueden aparecer negativos en algunas mquinas, y positivos en otras.

La ocurrencia mas comn de esta situaci¢n es cuando el valor -1 es usado para EOF. Considerar el código

```
char c;
c = getchar();
if (c == EOF)
    ...
```

En una mquina que no indica extensi¢n, c es siempre positivo porque este es un char, sin embargo EOF es negativo. Como resultado, la pregunta siempre falla. Para evitar esto, hemos sido ciudadosos para usar int en lugar de char para alguna variable que espera un valor devuelto por getchar.

La verdadera razón para usar int en lugar de char no esta relacionado con alguna pregunta de una posible indicaci¢n de extensi¢n. Es simplemente que getchar debe devolver todos los caracteres posibles (así que esto puede ser usado para leer entradas arbitrarias) y, en suma, un valor distinto de EOF. Así este valor no puede ser representado como un char, pero debe en cambio ser almacenado como un
int.

Otra forma til de conversi¢n de tipo automatico es que expresiones de relaci¢n como i > j y expresiones logicas conectadas por && y || son definidas por tener valor 1 si es verdadero, y 0 si es falsa. Así el asignamiento

```
isdigit = c >= '0' && c <= '9';
```

va a ser 1 si c es un d¡gito, y 0 si no lo es. (En la pregunta de un if, while, for, etc. "verdadera" significa "no-cero".)

En general, si un operador como + o * que toma dos operandos (un "operador binario") tiene operandos de diferentes tipos, el tipo "menor" es promovido al "mayor" antes de proceder a la operaci¢n. El resultado es del tipo "mayor". Mas precisamente, para cada operador aritmético, la siguiente secuencia de conversi¢n reglamenta esta aplicaci¢n.

char y short son convertidos a int, y float es convertido a double.

Entonces si un operando es double, el otro es convertido a double, y el resultado es double.

De otro modo si un operando es long, el otro es convertido a long, y el resultado es long.

De otro modo si un operando es unsigned, el otro es convertido a unsigned, y el resultado es unsigned.

De otro modo los operandos deben ser int, y el resultado es int.

Notar que todo float en una expresi¢n son convertidos a double; todo punto flotante aritmético en C es hecho en doble precisi¢n.

Las conversiones toman lugar a traves de asignamientos; el valor de la derecha es convertido al tipo de la izquierda, que es el tipo del resultado. Un carácter es convertido a un entero, por senal de extensi¢n como describimos antes.

La operaci¢n inversa, int a char, esta bien conducida - el exceso de un alto orden de bits son simplemente descartados. Así en

```
int i;
char c;
i = c;
c = i;
```

el valor de c no cambia. Esto es verdadero si la flag de extensi¢n esta involucrada o no.

Si x es float e i un int, entonces

```
x = i
```

y

```
i = x
```

ambas causan conversiones; float a int causa la truncaci¢n de alguna parte fraccional. double es convertido a float por redondeo. Los int's mas largos son convertidos a unos mas cortos o a char's por saltar el exceso de un alto orden de bits.

Ya que un argumento de funci¢n es una expresi¢n, los tipos de conversiones también toman lugar cuando los argumentos son pasados a funciones : en particular, char y short llegan a ser int, y float llega a ser double. Esto es porque tenemos declarados los argumentos de funci¢n para ser int y double aun cuando a funci¢n es llamada con char y
float.

Finalmente, tipos de conversiones explicitos pueden ser forzados en alguna expresi¢n con un constructo llamado cast. En la construcci¢n :

```
(tipo_de_nombre) expresi¢n
```

la "expresión" es convertida al tipo nombrado por las reglas de conversi¢n anterior. El significado preciso de un cast es, en efecto, como si "expresión" fuera asignada a una variable de tipo especificado, que es usada en lugar de la totalidad de las construcciones.

Por ejemplo, la rutina de biblioteca sqrt espera un argumento double.

Así, si n es un entero,

```
sqrt((double)n)
```

convierte n a double antes de pasar a sqrt. (Note que el cast produce el valor de n en el propio tipo; el contenido actual de n no es alterado.) El operador cast tiene la misma precedencia que los otros operadores unarios, como sumarizamos en la tabla al final de este **CAPITULO**.

2.8 OPERADORES DE INCREMENTO Y DECREMENTO

C provee dos inusuales operadores para incrementar y decrementar variables. El operador de incremento ++ suma su operando; el operador
de decremento -- sustrae uno. Hemos usado frecuentemente ++ para incrementar variables, como en

```
if (c == '\n')
    ++nl;
```

El aspecto inusual es que ++ y -- pueden ser usados como operadores prefijos (antes de la variable, como en ++n), o sufijo (después de la variable : n++). En ambos casos el efecto es el incremento de n. Pero la expresi¢n ++n incrementa n antes de usarse el valor, mientras que n++ incrementa n después que el valor ha sido usado. Esto significa que en un contexto donde el valor esta siendo usado, ++n y n++
son diferentes. Si n es 5, entonces

```
x = n++;
```

pone en x un 5, pero

```
x = ++n;
```

pone en x un 6. En ambos casos , n llega a ser 6. Los operadores de incremento y decremento pueden solo ser aplicados a variables; una expresi¢n como x = (i+j)++ es ilegal.

En un contexto donde el valor no es deseado, justo el efecto de incrementar, como en

```
if (c == '\n')
    nl++;
```

elegir prefijo o sufijo es cuestion de gusto. Pero hay situaciones donde una u otra es especificamente llamada. Por ejemplo, considerar la funci¢n squeeze(s,c) que traslada todas las ocurrencias del carácter c desde el string s.

```
squeeze(s,c)  /*  borra todo c desde s  */
char s[];
int c;
{
    int i,j;
    for (i = j = 0; s[i] != '\0'; i++)
        if (s[i] != c)
            s[j++] = s[i];
    s[j] = '\0';
}
```

Cada vez que ocurre un no-c, esto es copiado en la posici¢n j actual, y solo entonces j es incrementado para que este listo para el próximo carácter. Esto es exactamente equivalente a

```
if (s[i] != c)  {
    s[j] = s[i];
    j++;
}
```

Otro ejemplo de una construcci¢n similar viene de la funci¢n getline, donde podemos sustituir

```
if (c == '\n')  {
    s[i)]= c;
    ++i;
}
```

por el mas compacto

```
if (c == '\n')
    s [i++] = c;
```

Como un tercer ejemplo la funci¢n strcat(s,t) concatena el string t al final del string s. strcat asume que hay bastante espacio en s para la combinaci¢n.

```
strcat(s,t)     /*  concatena t al final de s
                    */
char s[], t[];  /*  s debe ser bastante grande  */
{

    int i, j;
    i = j = 0;

    while(s[i] != '\0')      /* encuentra fin de
                                s  */
        i++;
    while((s[i++] = t[j++]) != '\0')   /* copia
                                          t */
         ;

}
```

Como cada carácter es copiado desde t a s, el prefijo ++ es aplicado a i, j para estar seguro de que hay una posici¢n para el próximo paso que va hasta el final del loop.

2.9 OPERADORES LOGICOS BITWISE

C provee un n£mero de operadores para la manipulaci¢n de bit; estos

no debe ser aplicados a float o double.

```
&   bitwise AND
|   bitwise OR inclusivo
^   bitwise OR exclusivo
<<  cambio a la izquierda
>>  cambio a la derecha
~   complemento (unario)
```

El operador & es usado frecuentemente para desmascarar algn set de bits; por ejemplo,

```
c = n & 0177;
```

pone todos en cero, pero los 7 bits de mas bajo orden de n. El operador bitwise | es usado para activar bits:

```
x = x | MASK
```

pone un uno en x los bits que están puestos en uno en MASK.

Ud. debería distinguir cuidadosamente los operadores bitwise & y | de los conectivos logicos && y ||, que implican evaluaci¢n de derecha a izquierda de un valor verdadero. Por ejemplo, si x es 1 e y es 2, entonces x & y es cero mientras x&&y es uno. (porque?)

los operadores shift << y >> ejecutan cambios de izquierda a derecha de su operando izquierdo el n£mero de posiciones de bit dadas por el operando de la derecha. Así, x<<2 mueve x a la izquierda 2 posiciones, llenando los bits vacantes con 0; esto es equivalente a la multiplicaci¢n por 4. Corriendo a la derecha una cantidad unsigned llena los bits vacantes con 0. Corriendo a la derecha una cantidad indicada llenara con signos los bits ("shift aritmético") en algunas mquinas tales como el PDP-11, y con 0 bits ("shift logico") en otras.

El operador binario ~ yields los complementos de un entero; esto es, convierte cada l-esimo bit a cero y viceversa. Este operador se usa t¡picamente en expresiones como

```
x & ~077
```

que enmascara los ultimos 6 bits de x a cero. Note que x&~077 es independiente del largo de la palabra, es preferible así, por ejemplo x&0177700, asume que x es una cantidad de 16-bit. La forma mas facil no involucra un costo extra, ya que ~077 es una expresi¢n constante y es evaluada a tiempo compilado.

Para ilustrar el uso de algunos de los operadores de bit consideremos la funci¢n getbits(x,p,n) que devuelve (ajustado a la derecha) el n-esimo bit del campo de x que comienza en la posici¢n p. Asumimos que la posici¢n del bit 0 esta al extremo derecho y que n y p son valores positivos con sentido. Por ejemplo, getbits(x,4,3) devuelve los tres bits en las posiciones: 4, 3 y 2 ajustados a la derecha.

```
getbits(x,p,n)     /* obtiene n bits desde la posici¢n
                      p */
unsigned x, p, n
{
    return((x >> (p+1-n)) & ~(~0 << n));
}
```

x >> (p+1-n) mueve el campo deseado al extremo derecho de la palabra.

Declarando el argumento x como unsigned aseguramos que cuando esta
"movido a la derecha", los bits vacantes seran llenados con ceros, no los bits senalados, sin considerar la mquina, el programa se esta ejecutando. ~0 son todos los l-esimos bits; desplazandolo n posiciones a la izquierda con ~0<<n crea una mascara con ceros en los n bits de la derecha y unos en los otros; complementando con ~ fabrica una mascara con unos en los n bits de mas a la derecha.

2.10 OPERADORES Y EXPRESIONES DE ASIGNAMIENTO

Expresiones tales como

```
i = i + 2
```

en que el lado izquierdo esta repetido en la derecha puede ser escrito en la forma resumida

```
i += 2
```

usando un operador de asignamiento como +=.

Muchos operadores binarios tienen un correspondiente operador de asignamiento op=, donde op es uno de

```
+  -  *  /  %  <<  >>  &  ^  |
```

Si e1 y e2 son expresiones, entonces

```
e1 op= e2
```

es equivalente a

```
e1 = (e1) op (e2)
```

excepto que e1 es computado solo una vez. Notar el paréntesis alrededor de e2:

```
x *= y + 1
```

es actualmente

```
x = x * (y + 1)
```

mas bien que

```
x = x * y + 1
```

Como un ejemplo, la funci¢n bitcount cuenta el n£mero de l-bit en su argumento de enteros.

```
bitcount(n)   /*  cuenta l bits en n  */
unsigned n;
{
    int b;

    for (b = 0; n != 0 ; n >>= 1)
        if ( n & 01)
            b++;
    return(b);
}
```

Completamente aparte de la concisi¢n, los operadores de asignamiento tienen la ventaja que ellos corresponden mejor a lo que la gente piensa. Nosotros decimos "sumar 2 a i" o "incrementar i en 2", no "tomar i, sumarle 2, entonces ponga el resultado de vuelta en i". En suma, para expresiones complicadas como

```
yyva(yypv(p3+p4) + yypv(p1+p2)) += 2
```

el operador de asignamiento hace el código facil de entender, ya que el lector no ha de revisar laboriosamente que dos grandes expresiones son en efecto las mismas, o preguntarse porque ellas no lo son. Y un operador de asignamiento hasta puede ayudar al compilador para producir
un código mas eficiente.

Nosotros ya hemos usado el hecho que la instrucci¢n de asignamiento
tiene un valor y puede ocurrir en expresiones; el ejemplo mas comn es

```
while((c = getchar()) != EOF)
        ...
```

Los asignamientos pueden ocurrir también en expresiones usando los otros
operadores de asignamiento (+=, -=, etc.), aunque es menos frecuente que
ocurra.

El tipo de una expresi¢n de asignamiento es el tipo de su operando izquierdo.

2.11 EXPRESIONES CONDICIONALES

Las instrucciones

```
if (a > b)
    z = a;
else
    z = b;
```

en curso calculan en z el maximo de a y b. La expresi¢n condicional, escrita con el operador ternario ?:, provee una alternativa para escribir esto y construcciones similares. en la expresi¢n

```
e1 ? e2 : e3
```

la expresi¢n e1 es evaluada primero. Si no es cero (verdadero), entonces la expresi¢n e2 es evaluada, y ese es el valor de la expresi¢n condicional. De otro modo e3 es evaluado, y ese es el valor. Solo una de e2 y e3 es evaluada. Así en z queda el maximo de a y b,

```
z = (a > b) ? a : b ;     /* z = max(a,b)  */
```

Debería notarse que la expresi¢n condicional es en efecto una expresi¢n, y esta puede ser usada justo como alguna otra expresi¢n. Si e2 y e3 son de diferente tipo, el tipo del resultado es determinado por las reglas de conversi¢n discutidas tempranamente en este **CAPITULO**. Por ejemplo, si f es un float, y n es un int, entonces la expresi¢n

```
(n > 0) ? f : n
```

es de tipo double sin tomar en consideraci¢n si n es positivo o no.

Los paréntesis no son necesarios alrededor de la primera expresi¢n de una expresi¢n condicional, ya que la precedencia de ?: es muy baja, justo el asignamiento anterior. Ellos son aconsejables de cualquier modo, ya que ellas hacen que en la expresi¢n la condici¢n sea facil de ver.

La expresi¢n condicional frecuentemente dirige al código breve. Por ejemplo, este loop imprime n elementos de un arreglo, 10 por línea, con cada columna separada por un blanco, y con cada línea (incluyendo la
ultima) terminada exactamente por un newline.

```
for (i=0; i < n; i++)
    printf("%6d %c", a[i], (i%10==9 || i == n-1)
        ? '\n' : ' ');
```

Un newline es impreso después de cada decimo elemento, y después del enesimo. Todos los otros elementos son seguidos por un blanco. Aunque esto debería parecer falso, es instructivo intentar escribir esto sin la expresi¢n condicional.

2.12 PRECEDENCIA Y ORDEN DE EVALUACION

La tabla de mas abajo hace un sumario de las reglas para precedencia y asociatividad de todos los operadores, incluyendo aquellos
que no han sido aun discutidos. Los operadores en la misma línea tienen la misma precedencia; las filas están en orden de precedencia decreciente, así, por ejemplo, *, /, y % todos tiene la misma precedencia, que es mayor que la de + y -.

operador	asociatividad
() [] -> .	izqu. a der.
! ++ -- -(tipo) * & sizeof	der. a izqu.
* / %	izqu. a der.
+ -	izqu. a der.
<< >>	izqu. a der.
< <= > >=	izqu. a der.
== !=	izqu. a der.
&	izqu. a der.
^	izqu. a der.
|	izqu. a der.
&&	izqu. a der.
||	izqu. a der.
?:	der. a izqu.
= += -= etc.	der. a izqu.
, (cap.3)	izqu. a der.

Los operadores -> y . son usados para accesar miembros de estructuras; ellos seran cubiertos en el **CAPITULO** 6, con sizeof (largo de un objeto).

En el cap.5 discutiremos * (oblicuidad) y & (direcci¢n de).

Notar que la precedencia de los operadores logicos bitwise &, ^ y | caen mas bajo que == y !=. Esto implica que expresiones como :

```
if ((x & MASK) == 0) ...
```

debe estar enteramente entre paréntesis para entregar resultados apropiados.

Como mencionamos antes, las expresiones involucran uno de los operadores asociativos y conmutativos (*, +, &, ^ , |) pueden ser reordenados aun cuando están entre paréntesis. En muchos casos esto no
diferencia cualquier cosa; en situaciones donde debe, explicitar variables temporales que pueden ser usadas para forzar un orden particular de evaluaci¢n.

C, es semejante a muchos lenguajes, n especifica en que orden los operandos de un operador son evaluados. Por ejemplo, en una instrucci¢n como

```
x = f() + g();
```

f puede ser evaluada antes de g o viceversa; así si f o g alteran una variable externa de la cual los otros dependen, x puede depender del orden de evaluaci¢n. Nuevamente los resultados intermedios pueden ser
almacenados en variables temporales para asegurar una secuencia particular.

Similarmente, el orden en que los argumentos de una funci¢n son evaluados no esta especificado, así la instrucci¢n

```
printf("%d %d\n", ++n, power(2,n));  /* EQUIVOCADO */
```

puede (y hace) producir diferentes resultados en diferentes mquinas, dependiendo en si es incrementado o no antes que power sea llamada, la soluci¢n, en curso es escrita

```
++n;
printf("%d %d \n", n, power(2,n));
```

Los llamados de funci¢n anidan instrucciones de asignamiento e incrementan y decrementan operadores que causan "efectos laterales" - alguna variable cambiada como un producto de la evaluaci¢n de una expresi¢n. En alguna expresi¢n que involucre efectos laterales, alli pueden estar apoyos sutiles en el orden en que las variables toman parte en la expresi¢n almacenada. Una situaci¢n poco feliz es tipificada por la instrucci¢n

```
a [i] = i++;
```

La pregunta es si el subindice es el antiguo o el nuevo valor de i. El compilador puede hacer esto en diferentes formas, y generalmente diferentes respuestas dependiendo de la interpretaci¢n.

CAPITULO 3 CONTROL DE FLUJO

Las instrucciones de control de un lenguaje especifico es el orden en el que se hacen los computos.

/*Rolando = Corima*/

3.1 INSTRUCCIONES Y BLOCKS

Una expresión tal como x=0, i++ o printf("...") llega a ser una instrucción cuando es seguida por un punto y coma, como en :

```
x = 0;
i++;
printf("...");
```

En C, el punto y coma es el termino de una instrucción mas bien que un separador como en otros lenguajes.

Las llaves { y } son usadas para grupos de declaraciones e instrucciones dentro de un block de instrucciones, así que ellas son equivalentes a una sola instrucción.

Los paréntesis que rodean las instrucciones de una función son un ejemplo obvio; paréntesis alrededor de multiples instrucciones después de un if, else, while o for son otros. (Las variables pueden ser declaradas dentro de un block). Nunca hay un punto y coma después de
la llave derecha que pone fin a un block.

3.2 IF - ELSE

La instrucción if-else en C es usada para tomar decisiones. Formalmente, la sintaxis es :

```
if (expresión)
    instruccion-1
else
    instruccion-2
```

donde el else es opcional. La "expresión" es evaluada; si es verdadera (si "expresión" tiene un valor no-cero), la instruccion-1 es ejecutada. Si es falsa (la expresión es cero) y si hay un else la instruccion-2 es efectuada al instante.

Desde un if,simplemente se pregunta por el valor numérico de una expresión, lo mas obvio es escribiendo:

```
if (expresión)
```

en lugar de

```
if (expresión != 0)
```

Algunas veces esto es natural y claro; en otras veces esto esta escondido.

Porque el else de un if-else es opcional, hay una ambiguedad cuando un else es omitido desde una secuencia de if anidado. Este es resuelto en la manera usual, por ejemplo en :

```
if (n > 0)
    if (a > b)
        z = a;
    else
        z = b;
```

El else va con el if interno como lo muestra la identacion. Si esto no es lo que Ud. desea las llaves deben ser usadas para forzar la propia asociacion :

```
if (n > 0)  {
    if (a > b)
        z = a;
}
else
    z = b;
```

La ambiguedad es especialmente perniciosa en situaciones como :

```
if (n > 0)
    for (i = 0; i < n;i++)
        if (s[i] > 0)  {
        printf (...);
        return(i);
        }
        else
            printf("error-n es cero\n");
```

La identacion muestra inequivocamente que es lo que Ud. desea, pero
el compilador no entrega el mensaje y asocia el else con el if interno. Este tipo de error puede ser muy dificil de encontrar.

Por costumbre note que hay un punto y coma después de z=a en

```
if (a > b)
    z = a;
else
    z = b;
```

Esto es porque gramaticalmente, una instrucción que sigue al if, y una instrucción semejante a la expresión z=a es siempre terminada por un
punto y coma.

3.3 ELSE - IF

La construcción :

```
if (expresión)
    instrucción
else if (expresión)
        instrucción
    else if (expresión)
            instrucción
        else
            instrucción
```

ocurre a menudo; es equivalente a una breve discusion separada. Esta secuencia de if's es la manera mas general de escribir una multidecision. Las expresiones son evaluadas en orden, si alguna expresión es verdadera, la instrucción asociada con ella
es ejecutada, y esta termina toda la cadena. El código para cada instrucción es una instrucción cualquiera, o un grupo de ellas entre llaves.

El ultimo else dirige el "ninguna de las anteriores", u omite el caso cuando ninguna de las otras condiciones fue satisfecha. Algunas veces no hay una acción explicita para la omision; en ese caso

```
else
    instrucción
```

puede ser omitido, o puede ser usada para chequear un error o hallar una condición "imposible".

Para ilustrar una decisión triple, hay una función de busqueda binaria que decide si un particular valor de x se encuentra en el arreglo v, ordenado. Los elementos de v pueden estar en orden creciente. La función devuelve la posición (un numero entre 0 y n-1) si x se encuentra en v, y -1 si no.

```
binary(x,v,n) /* encuentra x en v[0]...v[n-1] */
int x, v[], n;
{
    int low, high, mid;
    low = 0;
    high = n-1;
    while (low <= high) {
        mid = (low + high ) / 2;
        if (x < v[mid])
            high = mid +1;
        else if (x > v[mid])
                low = mid +1;
            else   /* encuentra la pareja */
                return(mid);
    }
    return(-1);
}
```

La decisión fundamental es, si x es menor que, mayor que o igual al elemento central de v[mid] en cada incremento; esto es natural para else-if.

3.4 SWITCH

La instrucción switch es una multidecision especial, fabricante de preguntas, así una expresión que tiene un numero de expresiones constantes y bifurcaciones adecuadas. En el **CAPITULO** 1 escribimos un
programa que contaba las ocurrencias de cada dígito, espacios en blanco y otros caracteres, usando una secuencia de if...else if... else.

Aquí esta el mismo programa con un switch.

```
main () /* cuenta dígitos, espacios y otros */
{
    int c,i,nwhite,nother,ndigit[10];
    nwhite = nother = 0;
    for ( i= 0; i<10;i++)
        ndigit[i] = 0;
        while ((c = getchar()) != EOF)
            switch (c) {
            case '0' :
            case '1' :
            case '2' :
            case '3' :
            case '4' :
            case '5' :
            case '6' :
            case '7' :
            case '8' :
            case '9' :
            ndigit[c - '0']++;
            break;
            case ' ' :
            case '\n' :
            case '\t' :
            nwhite ++;
            break;
        default :
            nother++;
            break;
    }
    printf ("dígitos = ");
        for (i = 0; i < 10; i++)
            printf ("%d",ndigit[i]);
        printf ("\n espacios en blanco= %d, otros =
                %d\n",nwhite,nother);
}
```

Los switchs evaluan las expresiones enteras entre paréntesis (en este programa el carácter c) y compara su valor con todos los demás casos. Cada caso debe ser rotulado por un entero, carácter constante o expresión constante. Si un caso marca la expresión evaluada, la ejecucion parte desde ese caso. El caso rotulado default es ejecutado si ninguno de los otros casos es satisfecho. Un default es opcional; si este no esta alli, y ninguno de los casos aparejados, ninguna acción toma lugar. Los cases y default pueden suceder en algn orden. Los cases deben ser todos diferentes.

La instrucción break provoca una inmediata salida desde el switch porque los cases sirven justamente como rotulos, después que el código para un caso es hecho, la ejecucion desciende al próximo a menos que Ud.
tome acción explicita para escape.

break y return son las maneras mas comunes para abandonar switchs.

Una instrucción break también puede ser usada para forzar una salida inmediata de los ciclos while,for y do como será discutido mas tarde en este mismo **CAPITULO**.

En el lado positivo permite multiples casos para una simple acción, como son los blancos, TAB o newline, en este ejemplo.

3.5 CICLOS WHILE Y FOR

Ya nos hemos encontrado con los ciclos while y for. En

```
while (expresión)
    instrucción
```

"expresión" es evaluada si es no-cero, la "instrucción" es ejecutada y "expresión" es reevaluada. El ciclo continua hasta que "expresión" llegue a ser cero, en lo que indica que la ejecucion se reanuda después de "instrucción".

La instrucción for

```
for (exp1;exp2;exp3)
    instrucción
```

es equivalente a

```
exp1;
while (exp2) {
    instrucción
    exp3;
}
```

Gramaticalmente, las tres componentes de un for son expresiones. Mas comunmente, exp1 y exp3 son asignaciones o función de
llamadas y exp2 es una expresión relativa. Alguna de las partes puede

omitirse, aunque el punto y coma debe permanecer. Si la exp1 o exp3 es
omitida, es sencillamente desprendida de la expansion. Si la pregunta, exp2 no esta presente, es tomada como permanentemente verdadera, así

```
for ( ;; ) {
    ...
}
```

es un loop infinito, presumiblemente será interrumpido por otros medios tal como un break o un return.

Usar while o for es gran parte cuestion de gusto. Por ejemplo en

```
while ((c = getchar ()) == ' ' || c == '\n' ||
            c == '\t')
    ; /* salta caracteres con espacio en blanco */
```

no hay inicializacion o reinicializacion, así el while parece mas natural.

El for es claramente superior cuando no hay simple inicializacion y reinicializacion, ya que las instrucciones de control de lopp terminan a la vez y evidencian el tope del loop. Esto es mas obvio en

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

lo que es en idioma C para procesar los n primeros elementos de un arreglo, el an#logo del ciclo Do de Fortran o PL/1. La analogia no es perfecta, de cualqier modo, ya que el limite de un for puede ser alterado desde el interior del loop, y el control de la variable i retiene valores cuando el loop termina por alguna razón. Porque los componentes del for son expresiones arbitrarias; los ciclos for no están restringidos a progresiones aritméticas.

Como un gran ejemplo, hay otra versión de atoi para convertir en string a su equivalente numérico. Este es mas general; (en el **CAPITULO**
4 se muestra atof, lo cual hace la misma conversión para nmeros de punto flotante).

La estructura básica del programa refleja la forma de entrada :

```
saltar espacio en blanco, si hay alguno
obtiene senal, si hay alguna
obtiene parte entera, convertirlo
```

Cada incremento hace su parte, y deja las cosas en claro para el próximo paso. El conjunto de procesos termina en el primer carácter que no debe ser parte de un numero

```
atoi(s)   /* convierte s en entero  */
char[s];
{
    int i,n,sign;
    for(i=0;s[i]==' ' || s[i]=='\n' || s[i]=='\t';i++)
        ;  /* salta espacios en blanco  */
    sign = 1;
    if (s[i] == '+' || s[i] == '-')  /* senal  */
        sign = (s[i++] == '+') ? 1 : -1;
    for (n = i; s[i] >= '0' && s[i] <= '9'; i++)
        n = 10 * n + s[i] - '0';
    return (sign * n);
}
```

Las ventajas de tener los ciclos centralizados, son siempre mas obvios cuando hay varios loops anidados.

La siguiente función es shell para sortear un arreglo de enteros; la idea básica de shell es que en etapas proximas, elementos distintos son comparados, mejor que unos adyacentes, como en un simple intercambio
de ordenamientos. Esto tiende a eliminar grandes cantidades de desordenes rápidamente, así mas tarde, en las etapas hay menos trabajo que hacer. El intervalo entre los elementos comparados decrece gradualmente a uno, a cuyo punto el ordenamiento llega a ser un método de intercambio adyacente.

```
shell(v,n) /* ordena v[0]...v[n-1] en orden creciente */
int v[], n;
{
    int gap, i, j, temp;
    for (gap = n/2; gap > 0; gap /=2)
        for ( i = gap;i < n;i++)
          for (j = i-gap; j >= 0 && v[j] > v[j+gap];
                    j -= gap)  {
              temp = v[j];
              v[j] = v[j+gap];
              v[j+gap] = temp;
          }
}
```

Hay tres loops anidados. El loop del extremo controla el "gap" entre elementos comparados, retrocediendo desde n/2 por un factor de dos
pasos hasta que llegue a ser cero. El loop del medio compara cada par de elementos que esta separado por "gap"; el loop mas interno invierte algn elemento que este fuera de orden. Desde "gap", es eventualmente reducido a uno, todos los elementos son ordenados correctamente. Notar
que la generalidad de los for hacen encajar al loop externo en la misma forma que los otros, aun cuando esto no es una progresion aritmética.

Un operador final es la coma ",", la cual frecuentemente se usa en la instrucción for. Un par de expresiones de izquierda a derecha, el tipo y valor del resultado son del tipo y valor del operador derecho. Así en una instrucción for es posible colocar multiples expresiones en las diferentes partes; por ejemplo procesar dos indices paralelamente. Esto esta ilustrado en la función reverse(s), la cual invierte el orden del string s.

```
reverse(s)  /* invierte el orden del string S  */
char[s];
{
    int c, i, j;
    for (i = 0; j = strlen(s)-1; i < j; i++, j--) {
            c = s[i]; s[i] = s[j]; s[j] = c;
    }
}
```

Las comas que separan los argumentos de la función, variables en declaración, etc. no son operadores "coma", y no garantizan la evaluación de izquierda a derecha.

3.6 CICLOS DO-WHILE

El tercer loop en C, es el do-while, prueba el bottom después de hacer cada paso a traves del cuerpo del loop; el cuerpo es siempre ejecutado una vez, la sintaxis es :

```
do
    instrucción(es)
while (expresión);
```

"instrucción" es ejecutada, en seguida "expresión" es evaluada. Si es verdadera, "instrucción es evaluada nuevamente, si "expresión" es falsa, el loop termina.

Como podria esperarse, do-while es mucho menos usado que while y for.

La función itoa convierte un numero a un carácter de string (el inverso de atoi). El trabajo es un poco mas complicado de lo que pudo pensarse en un comienzo, porque los metodos simples de generar los dígitos, los genera en el orden equivocado. Hemos seleccionado al generador del string invertido.

```
itoa(n,s)  /* convierte n a caracteres en s  */
char[s];
int n;
{
    int i,sign;
    if (( sign = n) < 0 )  /* registra signo  */
        n = -n;  /* hace n positivo  */
    i = 0;
    do {       /* genera d~gitos en orden inverso  */
         s[i++] = n % 10 +'0';   /*  entrega próximo
                        dígito */

    } while ((n /= 10 ) > 0 );     /*lo borra*/
    if (sign < 0)
        s[i++] = '-';
        s[i] = '\0';
    reverse(s);
}
```

El do-while es necesario o al menos conveniente, desde que el menor carácter debe ser instalado en el arreglo s, desatendiendo el valor de n. También usamos paréntesis alradedor de la nica instrucción que completa el cuerpo del do-while, aunque ellos son innecesarios, así el apresurado lector no confundiria la parte del while con el principio de un loop while.

3.7 BREAK

Algunas veces es conveniente ser capaz de controlar la salida de un loop, con otras instrucciones.

La instrucción break proporciona una temprana salida del for, while y do, precisamente como desde el switch. Una instrucción break provoca
la inmediata salida del loop mas interno.

El siguiente programa remueve, arrastrando blancos y tabs desde el final de cada línea de entrada, usando un break para salir de un loop cuando el no-blanco, no-tab es encontrado mas a la derecha.

```
#define MAXLINE  1000
main ()  /*  remueve arrastrando blancos y tabs  */
{
    int n;
    char line[MAXLINE];
    while (( n = getline(line,MAXLINE)) > 0) {
        while (--n >= 0)
            if (line[n] != ' ' && line[n] != '\t'
                        && line[n] != '\n')

                break;
            line[n+1] = '\0';
        printf("%s\n",line)
    }
}
```

getline devuelve el largo de la línea. El while interno comienza en el ultimo carácter de line, y examina al reves mirando al primer carácter que no es un blanco, TAB o newline. El loop es interrumpido cuando uno es encontrado, o cuando n es negativo (esto es, cuando la línea entera ha sido examinada).

Ud. debería verificar que esto es correcto al proceder, cuando la línea contiene caracteres en blanco. Una alternativa para break es poner la condición en el loop mismo:

```
while((n = getline(line, MAXLINE)) > 0) {
    while(--n >= 0
        && (line[n] == ' ' || line[n] == '\t'
        || line[n] == '\n'))
        ;
        ...
}
```

Esto es inferior a la versión previa, porque la pregunta es dificil de entender. Las preguntas que requieran una mezcla de &&, ||, ! o paréntesis deberian generalmente ser evitadas.

3.8 CONTINUE

La intruccion continue esta relacionada con break, pero menos frecuentemente usada; provoca que la próxima iteracion del loop(for, while, do) comience. En el while y do, esto significa que la parte de la pregunta es ejecutada inmediatamente en el for, controla el paso a la reinicializacion del step. (continue se aplica solo a loops, no a switch. Un continue interior a un switch, interior a un loop, provoca la próxima iteracion del loop).

Como un ejemplo, este fragmento procesa solamente elementos positivos en el arreglo a; los valores negativos son saltados.

```
for(i = 0;i < N;i++) {
    if( a[i] < 0)  /* se salta los negativos */
        continue;
        ...  /* hace elementos positivos */
}
```

La instrucción continue es usada frecuentemente cuando la parte del loop que sigue es complicada.

3.9 GOTO'S Y LABELS

C proporciona la infinitamente til instrucción goto, y rotulos para bifurcaciones. Formalmente el goto nunca es necesario, y en la practica es siempre facil para escribir códigos sin el.

Sugeririamos unas pocas situaciones donde los goto's tendrian lugar. El uso mas comn es el abandono del proceso en alguna estructura anidada profundamente, (tales como la ruptura de los loops a la vez). La instrucción break no puede ser usada directamente desde que deja solo el loop mas interno.

Así :

```
for (...)
    for (...)  {
        ...
        if( desastre)
            goto error;
    }
 ...
error :
    borrar - mensaje
```

Esta organizacion es socorrida si el manejo del código de error es no-trivial, y si el error puede ocurrir en varios lugares. Un rotulo tiene la misma forma como un nombre de variable, y es regida por una coma. Puede ser anexo a alguna instrucción como el goto.

Como otro ejemplo, considere el problema de encontrar el primer elemento negativo en un arreglo bidimensional (cap.5).

Una posibilidad es :

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

```
            if(v[i][j] < 0)
                goto found;
            /* no lo encuentra */
            ...
        found:
           /* encuentra uno en la posición i,j */
        ...
```

El código implica que un goto puede siempre ser escrito. Recuerde que es comn usarlo para desviar la secuencia hacia algn menzaje de error. Aunque quizas el precio de algunas preguntas repetidas o una variable extra. Por elemplo el arreglo de busqueda llega a ser.

```
found = 0;
for (i = 0; i < N && !found; i++)
    for (j = 0; j < M && !found; j++)
        found = v[i][j] < 0;
if (found)
    /*  el elemento estuvo en i-1, j-1  */

    ...
else
    /*  no encontrado  */
    ...
```

Aunque no somos dogmaticos sobre la materia, esto hace parecer que las instrucciones goto serian usadas raramente.

CAPITULO 4 FUNCIONES Y ESTRUCTURA DE PROGRAMAS

C ha sido construido para fabricar funciones eficientes y faciles de usar; los programas en C, generalmente consisten de numerosas funciones reducidas. Un programa puede residir en uno o mas archivos fuente en alguna forma conveniente; los archivos fuente pueden ser compilados separadamente y llamados juntos, con funciones compiladas previamente desde bibliotecas.

Muchos programadores están familiarizados con "funciones de biblioteca" para I/O (getchar, putchar) y calculos numericos (sin, cos, sqrt). En este **CAPITULO** mostraremos mas acerca de escribir nuevas funciones.

4.1 BASICS

Para comenzar, construiremos y escribiremos un programa para imprimir cada línea de su entrada que contenga un modelo particular o string de caracteres. (Este es un caso especial del UNIX, programa utilitario grep). Por ejemplo, buscando por el modelo "the" en el set de líneas :

```
Now is the time
for all good
men to come to the aid
of their party
```

producira la salida

```
Now is the time
men to come to the aid
of their party
```

La estructura básica del job cae netamente en tres partes :

```
while ( hay otra línea? )
    if ( la línea contiene el modelo? )
        imprimalo
```

Aunque es ciertamente posible poner el código para todos estos en la rutina principal, una mejor manera es usar la estructura natural para mejorar cada parte como una función separada.

"hay otra línea?" es getline, una función que ecribimos en el **CAPITULO** 1 e "imprimalo" es printf. Esto significa que necesitamos escribir una rutina que decida si la línea contiene una ocurrencia del modelo. Nosotros podemos resolver este problema "robando" una construcción de PL/1 : la funcio}n index(s,t) que devuelve la posición o indice en el string s donde el string t comienza, o -1 si s no contiene a t.

Nosotros usaremos 0 en vez de 1 como la posición inicial en s porque los arreglos en C comienzan de la posición cero. Cuando mas tarde necesitemos modelos mas sofisticados solo tenemos que reemplazar
index; el resto del código puede permanecer, el mismo.

Por ahora, el modelo a buscar es un string literal en el argumento de index, que no es el mas general de los mecanismos. Volveremos a retomar este asunto para discutir como inicializar arreglos de carácter. Esta es una nueva versión del getline; Ud. debe encontrarla instructiva para compararla con la del **CAPITULO** 1.

```
#define MAXLINE 1000
main ()           /* encuentra las líneas que
{                    contienen el modelo     */

    char line[MAXLINE];
    while (getline(line, MAXLINE) > 0)
        if (index(line,"the") >= 0)
            printf("%s",line);
}

getline(s,lim)          /*entrega línea en s, devuelve
                  el largo            */
char s[];
int lim;
{
   int c,i;
    i = 0;

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

```
index(s,t)   /*devuelve  el indice de t en s,
                -1 si ninguno  */
char s[],t[];
{
    int i, j, k;
    for (i=0;s[i] != '\0';i++) {
      for (j=1,k=0;t[k] !='\0' && s[j] == t[k];j++,k++)
            ;
     if (t[k] == '\0')
        return(i);
    }
  return(-1);
}
```

Cada función tiene la forma

```
nombre (lista de argum., si hay alguno)
declaración de argumentos, si hay algunos
{
    declaraciones e instrucciones, si hay
}
```

Como sugerencia, las diferentes partes pueden estar ausentes; una minima función es

```
dummy()
```

la que no hace nada.(una función que no hace nada es til a veces como un lugar de espera durante el desarrollo del programa.) El nombre de la función puede ser precedido también por un type si la función devuelve alguna otra cosa en vez de un valor entero; este es el tipico de la próxima seccion.

Un programa es justamente un conjunto de funciones definidas individualmente. La comunicación entre las funciones es (en este caso) por argumentos y valores devueltos por las funciones; también puede ser via variables externas. Las funciones pueden ocurrir en algn orden en el archivo fuente.

4.2 FUNCIONES QUE DEVUELVEN VALORES NO ENTEROS.

Escribamos la función atof(s), la cual convierte el string s a su equivalente punto-flotante doble precisión. atof es una extension del atoi, versiones escritas en los **CAPITULO**s 2 y 3.

Primero, atof, por si mismo debe declarar el tipo de valor a retomar, desde no-entero. Porque float es convertido a double en las expresiones, no hay un punto para decir que atof devuelve float;debimos hacer uso de la precisión extra y así lo declaramos para que devuelva un valor double. El tipo de nombre precede el numero de la función, semejante a esto :

```
double atof(s)  /* convierte string s a double */
char s  ;

    double  val, power;
    int i, sign;

    for(i=0;s i ==' '[[s i ==' n'[[ s i ==' t';i++)
          ;       /* salta espacio en blanco */

    sign = 1;
    if (s i  == '+' [[ s i  == '-')   /* se|al */
        sign = (s i++ =='+') ? 1 : -1;
    for (val=0; s i  >= '0' && s i  <= '9';i++)
        val =10 * val + s i  - '0';
    if (s i  == '.')
        i++;
    for ( power =1;s i  >= '0' && s i  <= '9'; i++)
        val =10 * val + s i  - '0';
        power *=10;

    return(sign * val / power);
```

Segundo, y muy importante la rutina de llamada debe mencionar que atof devuelve un valor no-int. La declaración es mostrada en la siguiente calculadora de escritorio, primitiva (adecuada para balances de chequeos de libros), el cual lee un numero por línea, opcionalmente precedido de un signo, y sumados todos ellos imprimiendo la suma después
de cada entrada.

```
define MAXLINE 100

main()  /* calculadora de escritorio, rudimentaria */

    double sum, atof();
    char line MAXLINE ;

    sum = 0;
    while (getline(line, MAXLINE) > 0)
        printf (" t%.2f n",sum += atof(line));
```

La declaración

```
double sum,atof();
```

dice que sum es una variable double, y que atof es una función que devuelve un valor double. Como una nemotecnica, sugerimos que sum y
atof(...) son ambos valores de punto flotante de doble precisión.

A menos que atof este explicitamente declarado en ambos lugares, C asume que devuelve un entero, y entregara respuestas sin sentido. Si atof mismo y la llamada para el, en main, son tipeados incompatiblemente
en el mismo archivo de origen, será detectado por el compilador. Pero si (como es mas probable) atof fue compilado separadamente, la separacion no seria detectada,atof devolveria un double, que main tratara como un int, y resultarian respuestas insensatas

Mencionado atof,podriamos en principio escribir atoi (convertir un string a int) en terminos de el:

```
atoi(s)       /* convierte string a entero */
char s ;

    double atof();
    return(atof(s));
```

Notar la estructura de las declaraciones en la instrucción return. El valor de la expresión en

```
return("expresión")
```

es siempre convertido al tipo de la función antes de que el return sea tomado. Por esto, el valor de atof, a double, es convertido automáticamente a int cuando aparece en un return, desde la función atoi
devuelve un int (la conversión de un valor de punto flotante a int trunca alguna parte fraccional, como discutimos en el **CAPITULO** 2).

4.3 MAS SOBRE ARGUMENTOS DE FUNCIONES

En el cap.1 discutimos el hecho que los argumentos de función son pasados a valor, esto es,la función llamada acepta uno particular, copia temporal de cada argumento, no sus direcciones. Esto significa que la función no puede afectar al argumento original en la llamada de la función. Dentro de una función, cada argumento es en efecto una variable local inicializada por el valor con el cual la función fue llamada.

Cuando un nombre de arreglo aparece como un argumento para una función, la ubicacion del comienzo del arreglo es pasada; los elementos no son copiados. La función puede alterar elementos del arreglo por subscripcion desde esta ubicacion. El efecto es que los arreglos son pasados por referencia. En el cap.5 discutiremos el uso de punteros para permitir funciones que afecten no-arreglos en funciones llamadas.

Por la forma, no hay una manera integramente satisfactoria para escribir una función portatil que acepte un numero variable de argumentos, porque no hay forma portatil para una función llamada, determinar cuantos argumentos fueron actualmente transferidos a ella en una llamada dada. Así, Ud. no puede escribir una función verdaderamente portatil que complete el maximo de un numero arbitrario
de argumentos, cm seria la función MAX de Fortran o PL/1.

Est es generalmente seguro para tratar con un numero variable de argumentos. Si la función llamada no usa un argumento que no fueron actualmente abastecidas, y si los tipos son consistentes. printf, la función mas comn en C, con un numero variable de argumentos, utiliza informacion desde el primer argumento para determinar cuantos argumentos
están presentes y de que tipo son. Lo deja si el que llama no sumunistra bastantes argumentos, o si los tipos no son los que el primer argumento dijo. Esto es también no portatil y debe ser modificado para diferentes desarrollos.

Alternativamente, si los argumentos son de tipo conocido es posible marcar el final de la lista de argumentos en algune forma convenida, tal como un especial valor de argumentos (frecuentemente cero) que representa el fin de los argumentos.

4.4 VARIABLES EXTERNAS

Un programa C consiste en sets de objetos externos, que son variables y funciones. El adjetivo "external" es usado primariamente en contraste a "internal", que describe los argumentos y variables automaticas definidas dentro de funciones. Las variables externas son definidas fuera de alguna función, y son así aprovechables por otras funciones. Las funciones mismas son siempre externas, porque C no permite definir funciones dentro de otras funciones. Por omision de variables externas son también "global", así que todas las referencias a tal, una variable por el mismo nombre (aun desde funciones compiladas separadamente) son referenciadas por la misma cosa. En este sentido, laas variables externas son analogas al Fortran COMMON o PL/1 EXTERNAL.
Veremos mas tarde como definir variables externas y funciones que no son
globalmente aprovechables, pero son en vez de visibles solo dentro de un
simple archivo fuente.

Porque las variables externas son globalmente accesibles, ellas proporcionan una alternativa para argumentos de funciones y devuelven valores para comunicar datos entre funciones. Alguna función puede accesar una variable externa refiriendose a su nombre, si el nombre ha sido declarado de algn modo.

Si un gran numero de variables debe ser compartido entre varias funciones las variables externas son mas convenientes y eficientes que una larga lista de argumentos. Este razonamiento debería ser aplicado con alguna precaucion, esto puede tener un mal efecto en la estructura del programa, y dirigir programas con muchas conexiones entre funciones.

Una segunda razón para usar variables externas concierne a la inicializacion. En particular, los arreglos externos pueden ser inicializados, pero los arreglos automaticos no. Trataremos la inicializacion al final de este **CAPITULO**.

La tercera razón para usar variables externas es su alcance y "eternidad". Las variables automaticas son internas a una función; ellas vienen a existir cuando la rutina esta entera, y desaparece cuando esta a la izquierda. Las variables externas son permanentes. Ellas no vienen y van, retienen valores desde una invocacion de función a la próxima. De este modo, si dos funciones deben compartir algunos datos,
ninguna llama a la otra, frecuentemente es mas conveniente si el dato compartido es guardado en variables externas mejor que pasarlos via argumentos.

Examinaremos esta edicion mas tarde con un gran ejemplo. El programa es escribir otro programa calculadora, mejor que el anterior. Este permite +, -, *, / y = (para imprimir la respuesta). Poque es algo facil para implementar, la calculadora usara notación polaca inversa en vez de infix. En la notación polaca inversa, cada operador sigue su operando; una expresión infix como :

```
(1 - 2) * (4 + 5) =
```

es entrada como

```
1 2 - 4 5 + * =
```

Los paréntesis no son necesarios.

La implementacion es absolutamente simple. Cada operando es puesto en un stack; cuando un operador llega, los propios nmeros de operandos (dos para operadores binarios) son soltados, el operando aplicado a ellos, y el resultado vuelto a poner en el stack. En el ejemplo anterior, 1 y 2 son puestos, entonces reemplazados por su diferencia, -1. Siguiendo 4 y 5 sonpuestos y reemplazados por su suma, 9. El producto de -1 y 9, que es
-9, los reemplaza en el stack. El operador = imprime el elemento tope sin removerlo (así los pasos inmediatos en un calculo pueden ser chequeados).

Las operaciones de entrada y salida de un stack son triviales, pero por la deteccion y recuperacion del tiempo de error son sumados, ellos son bastante largos, es mejor poner cada uno en una función que repetir completamente el código del programa entero. Y alli debería ser una función separada para traer la próxima entrada de operador u operando. Así la estructura del programa es :

```
while (próximo operador u operando si no es EOF)
    if (numero)
        pongalo
    else if (operador)
        soltar operandos
        hacer operación
        poner resultado
    else
        error
```

El diseno principal de decisión que no ha sido aun discutido es donde esta el stack, esto es, que rutinas lo accesan directamente. Una posibilidad es guardarlo en main, y pasar el stack y la posición del stack en curso a las rutinas que ingresen y saquen informacion de el. Pero main no necesita conocer las variables que controlan el stack; se debería pensar solo en terminos de ingresar y sacar. Así hemos decidido fabricar el stack y asociar la informacion de variables externas accesible a las funciones push y pop pero no es main. El programa principal es primeramente un gran witch en el tipo de operador
u operando; esto es quizas el mas tipico uso de switch que el mostrado

en el cap.3.

```
define MAXOP   20  /* largo maximo de operando y
                      operador */
define NUMBER  '0'  /* senal que el numero ha sido
                       encontrado  */
define TOOBIG  '9'  /* se|al que el string es
                 demasiado grande */
main()   /* calculadora de escritorio, notación polaca
                            inversa  */

    int type;
    char s MAXOP ;
    double op2, atof(), pop(), push();

    while ((type = getop(s, MAXOP)) != EOF)
        switch (type)

        case NUMBER :
            push(atof(s));
            break;
        case '+' :
            push(pop() + pop());
            break;
        case '*' :
            push(pop() * pop());
            break;
        case '-' :
            op2 = pop();
            push(pop() - op2);
            break;
        case '/' :
            op2 = pop();
            if (op2 != 0.0)
                push(pop() / op2);
            else
                printf("división por cero n");
            break;
        case '=' :
            printf(" t%f n",push(pop()));
            break;
        case 'c' :
            clear();
            break;
        case TOOBIG :
            printf("%.02s...es demasiado largo
                        n",s);
            break;
        default :
            printf("comando desconocido %c n",type);
            break;



define MAXVAL  100   /* fondo maximo del stack  */

int sp = 0;          /* puntero del stack  */
double push(f)       /* push f onto value stack  */
double f;

    if (sp < MAXVAL)
        return(val sp++  = f);
    else
        printf("error : stack lleno n");
        clear();
        return(0);

double pop()  /* suelta el tope del stack  */

        if (sp > 0)
            return (val --sp );
        else
            printf("error : stack vacío n");
            clear();
            return(0);


clear()       /*  limpia el stack  */

    sp = 0;
```

El comando c limpia el stack, con una función que es también usada por push y pop en el caso de error, Volveremoe con getop en un momento.

Como discutimos en el cap.1, una variable es externa si es definida fuera del cuerpo de alguna función. Así el stack y el puntero del stack que debe ser compartido por push, pop, y clear son definidas fuera de estas tres funciones. Pero main no se refiere al stack o al puntero - la representación es ocultada cuidadosamente. Así el c}digo para el operador = debe usar

```
push(pop());
```

para examinar el tope del stack sin alterarlo.

Notar tambi`n que, porque + y * son operadores conmutativos, el orden en el que los operandos soltados son combinados es irrelevante, pero para los operadores - y / los operandos de izquierda y derecha deben ser distinguidos.

4.5 ALCANCE DE LAS REGLAS

las funciones y variables externas que completen un programa C no necesitan ser compiladas, todas al mismo tiempo; el texto del programa puede ser guardado en varios archivos y rutinas previamente compiladas pueden ser llamadas desde biblioteacs. las dos preguntas de inter,s son :

- ¨ como son escritas las declaraciones, de suerte que las variables son declaradas propiamente durante la compilacion ?
- ¨ como fijar declaraciones de suerte que todas las partes ser#n conectadas propiamente cuando el programa es llamado ?

El alcance de un nombre es la parte del programa sobre la cual el nombre esta definido. Para una variable automatica declarada al comienzo de una función, el alcance es la función en la que el nombre es
declarado, y las variables del mismo nombre en diferentes funciones no tienen relacion. Lo mismo es verdadero para los argumentos de la función.

El alcance que una variable externa tiene es; desde el punto en que es declarada en un archivo fuente hasta el final de ese archivo. Por ejemplo, si val, sp, push, pop y clear son definidas en un archivo en el orden mostrado anteriormente, esto es,

```
int sp = 0;
double val MAXVAL ;
double push(f)  ...
double pop()    ...
clear()   ...
```

entonces las variables val y sp pueden ser usadas en push, pop y clear simplemente nombrandolas; las declaraciones no son necesitadas mas alla.

Si una variable externa es referida antes, esta es definida, o si es definida en un archivo fuente diferente desde uno donde est siendo usado, entonces es obligatoria una declaración extern.

Es importante distinguir entre la declaración de una variable externa y su definición. Una declaración anuncia las propiedades de

una variable (su tipo, largo, etc.); una definici¢n tambi,n provoca el almacenaje para ser asignada. Si las líneas

```
int sp;
double val MAXVAL ;
```

aparece fuera de alguna función, ella definen las variables externas sp y val, provoca el almacenaje para ser asignada, y también sirve como la declaración para el resto del archivo fuente. Las líneas

```
extern int sp;
extern double val  ;
```

declaran para el resto del archivo fuente que sp es un int y que val es un arreglo double (cuyo largo es determinado en otra parte), pero ellas no crean las variables o les asignan almecenaje.

Allí debe estar solo una definición de una variable externa entre todos los archivos que completan el programa fuente; otros archivos pueden contener declaraciones extern para declararlo. (Allí también puede estar una declaración extern en el archivo conteniendo una definición.) Alguna inicializacion de una variable externa va solo con la definición. Los largos de los arreglos deben ser especificados con la definición, pero son opcionales con una declaración extern.

Aunque no es probable una organizacion para este programa, val y sp pudieron ser definidas e inicializadas en un archivo, y las funciones push, pop y clear definidas en otro. Entonces estas definiciones y declaraciones deberian ser necesarias para enlazarlas juntas :

en archivo1 :

```
int sp = 0;  /* puntero del stack */
double val MAXVAL ;  /*  valor del stack */
```

en archivo2 :

```
extern int sp;
extern double val  ;
double push(f)  ...
double pop ()  ...
clear()  ...
```

Porque las declaraciones en el archivo 2 están situadas a la cabeza y fuera de las tres funciones, ellas aplican a todas un conjunto de declaraciones son suficientes para todo el archivo 2.

Para programas largos, el archivo de inclucion define facilita la discusion mas tarde en este **CAPITULO** permitiremos una para guardar solo
una simple copia de las declaraciones extern por el programa y tiene que insertar en cada archivo fuente como est# siendo compilado.

Volvamos ahora a la implementacion de getop, la función que va a buscar el próximo operador u operando. La tarea básica es facil: salta blancos,tabs y newlines. Si el próximo carácter no es un dígito o un punto decimal, vuelve. De otra manera, colecciona un string de dígitos (que deben incluir un punto decimal), y devuelve NUMBER, la se|al que el
numero ha sido recogido.

La rutina es substancialmente complicada para intentar manejar la situaci¢n propiamente cuando un n£mero de entrada es demasiado largo. getop lee d¡gitos (quizas con una intervenci¢n del punto decimal) hasta no ver alguno ms, pero s¢lo almacanan los que sean adecuados. Si no hubo overflow, devuelve NUMBER y el string de d¡gitos. Si el n£mero fu, demasiado largo, de cualquier modo, getop descarta el resto de la línea de entrada, as¡ el usuario puede simplemente retipear la línea desde el punto de error; devuelve TOOBIG como se¤al de overflow.

```
getop(s,lim)   /*  entrega pr}ximo operador u operando
                                */
char s  ;
int lim;
int i, c;

while ((c =getch()) == ' ' !! c == ' t'
                !! c == ' n')
            ;
if (c != '.' && (c < '0' !! c > '9'))
    return(c);
s 0  = c;
for (i=1;(c=getchar()) >= '0' && c <= '9';i++)
    if (i < lim)
        s i  = c;
if (c == '.')    /*  fracci¢n de la colecci¢n  */
    if ( i < lim)
        s i  = c;
    for (i++;(c = getchar()) >= '0' && c <= '9';
                            i++)
        if (i < lim)
            s i  =c;


if (i < lim)    /* el n£mero est listo */
    ungetch(c);
    s i  = ' 0';
    return(NUMBER);
  else      /* esto es demasiado grande; salta
                    el resto de la línea */
     while (c != ' n' && c != EOF)
        c = getchar();
     s lim-1   = ' 0';
     return(TOOBIG);
```

¨ Qu, son getch y ungetch ? Frecuentemente es el caso que un programa leyendo la netrada no pueda determinar que ha leído lo suficiente hasta que ha leído demasiado. Una instancia es reciger los caracteres que completan un n£mero : hasta que el primer no-d¡gito es visto, el n£mero no est completo. Pero entonces el programa ha leído un carácter demaciado lejano, un carácter que no est# preparado.

El problema ser¡a resuelto si fuera posible no-leer el carácter no deseado. Entonces, cada vez que el programa lee demaciados caracteres,
lo pudo regresar a la entrada, as¡ el resto del c¢digo pudo proceder como si nuca fu, leído. Afortunadamente, es fcil simular perder un carácter, escribiendo un par de funciones de cooperaci¢n. getch libera el pr¢ximo carácter a ser considerado; ungetch pone un carácter de regreso en la entrada, as¡ que el pr¢ximo llamado para getch lo devolver meramente.

Como ellas trabajan juntas, es simple. ungetch pone los caracteres "devueltos" en un buffer compartido - un arreglo de caracteres. getch lee desde el buffer si hay alguna cosa all¡; lo llama getchar si el buffer est vac¡o. All¡ deber¡a estar tambi,n un ¡ndice variable que registra la posici¢n del carácter en curso en el buffer.

Desde el buffer y el ¡ndice son compartidos por getch y ungetch,y deben retener sus valores entre las llamdas, ellas deben ser externas a ambas rutinas. As¡ podemos escribir getch, ungetch, y sus variables compartidas como :

```
define BUFSIZE  100

char buf BUFSIZE  ;   /*  buffer para ungetch  */
int bufp = 0;        /* pr}xima posici}n libre en el
                    buffer  */

getch()             /*  entrega un (posiblemente de
                vuelta) carácter  */
```

```
    return((bufp > 0) ? buf --bufp   : getchar());


ungetch(c)          /* pone carácter de vuelta en la
                    entrada */
int c;


    if (bufp > BUFSIZE)
        printf("ungetch : demasiados caracteres n");
    else
        buf bufp++  = c;
```

Hemos usado un arreglo para los devueltos, mejor que un simple carácter,
ya que la generalidad puede entrar manualmente, mas tarde.

4.6 VARIABLES ESTATICAS

Las variables estticas son una tercera clase de almacenaje, en suma a las extern y automticas que ya hemos encontrado.

Las variables estticas pueden ser internas o externas. Las variables internas son locales a una funci¢n particular justo como son las variables automticas, ellas permanecen en existencia, mejor dicho que vienen y van cada vez que la funci¢n es activada. Esto significa que las variables static internas proporcionan en forma privada un permanente almecenaje en una funci¢n. Los string de caracteres que aparecen dentro de una funci¢n, tales como los argumentos de printf son static internas.

Una variable static externa es conocida dentro del archivo fuente en el que es declarado, pero no en alg£n otro archivo. As¡ las variables static externas proporcionan una forma para ocultar nombres como buf y bufp en la combinaci¢n getch-ungetch, la que debe ser externa, as¡ ellas pueden ser compartidas, pero las que no deber¡an ser visibles para los usuarios de getch y ungetch, as¡ no hay posibilidad de conflicto. Si las dos rutinas y las dos variables son compiladas en un archivo, como

```
    static char   buf BUFSIZE ;
    static int    bufp = 0;

    getch()  ...
    ungetch(c)  ...
```

entonces otra rutina no ser# capaz de accesar buf y bufp; en efecto, ellas no desean conflicto con los mismos nombres en otros archivos del mismo programa.

El almacenaje esttico, sea interno o externo, es especificado prefijando la declaraci¢n normal con la palabra static. La variable es externa si est definida fuera de una funci¢n, e interna si est definida dentro de la funci¢n.

Normalmente las funciones son objetos externos; sus nombres son conocidos globalmente. Es posible, de cualquier modo, para una funci¢n
ser declarada static; esto hace su nombre desconocido fuera del archivo en que es declarada.

En C, static connota no s¢lo permanencia sino tambi,n un grado que debe ser llamado "reserva". Los objetos internos static son conocidos s¢lo dentro de una funci¢n; los objetos static externos (variables o funciones) son conocidos s¢lo dentro del archivo fuente en que aparecen,
y sus nombres no interfieren con con variables del mismo nombre en otros
archivos.

Las variables static externas y funciones proporcionan una forma para ocultar datos, objetos y algunas rutinas internas que las manipulan, no pueden chocar inadvertidamente. Por ejemplo, getch y ungetch forman un "m¢dulo" para carácter de I/O; buf y bufp deber¡an ser
static as¡ ellas son inaccesibles desde fuera. En la misma forma, push, pop y clear forman un m¢dulo para la manipulaci¢n del stack; val y
sp deber¡an ser tambi,n static externas.

4.7 VARIABLES REGISTRO

La cuarta y £ltima clase de almacenaje es llamada register. Una declaraci¢n register avisa al compilador que la variable en cuesti¢n sea usada "pesadamente". Cuando las posibles variables register son ubicadas en mquinas registradas que pueden resultar en peque¤os y rpidos programas.

La declaración register tiene el efecto .

```
    register int   x;
    register char  c;
```

y as¡ la parte int puede ser omitida. register puede ser aplicado s¢lo a variables automticas y a los parmetros formales de una funci¢n. En este £ltimo caso la declaraci¢n tiene el efecto

```
    f(c,n)
    register int c,n;


        register int i;
        ...
```

En la prctica, hay algunas restricciones en variables register, reflejando las realidades del hardware fundamental. S¢lo unas pocas variables en cada funci¢n puede ser guardada en registros, y s¢lo ciertos tipos son permitidos. La palabra register es ignorada por exceso o rechazo de declaraciones. Y no es posibles tomar la direcci¢n de una variable register (este t¡pico lo cubriremos en el cap.5). Las restricciones especificas var¡an de mquina a mquina; como ejemplo, en el
PDP-11, s¢lo las primeras tres declaraciones register son efectivas, y el tipo debe ser int, char o puntero.

4.8 ESTRUCTURA DE BLOQUE

C no es un lenguaje estructurado en bloque en el sentido de PL/1 o Algol, en que las funciones pueden ser declaradas dentro de otras funciones.

Sin embargo, las variables pueden ser definidas en una forma de estructura de bloque. Las declaraciones de variables (incluyendo inicializadores) pueden seguir el par,ntesis izquierdo que introduce alguna combinaci}n de instrucciones, no justamente el que comienza una
funci¢n. Las variables declaradas en esta forma reemplazan id,nticamente algunas variables nombradas en otros bloques, y permanecen
en existencia hasta el par,ntesis derecho. Por ejemplo, en

```
    if(n > 0)
        int i;   /* declara un nuevo i  */
        for (i = 0; i < n; i++)
            ...
```

el alcance de la variable i es verdadero, bifurca al if; esta i no est relacionada a alguna otra i en el programa.

Los bloques de estructuras se aplican tambi,n a las variables externas. Dadas las declaraciones

```
int x;

f()

    double x;
    ...
```

entonces dentro de la funci¢n f, las ocurrencias de x se refiren a la variable interna doble; fuera de f, se refieren al entero externo. Lo mismo es verdad para nombres de par,ntesis formales

```
int z;

f(z)
double z;

    ...
```

Dentro de la funci¢n f, z se refiere al parmetro formal, no al externo.

4.9 INICIALIZACION

Esta secci¢n sumariza algunas de las reglas, ahora que hemos discutido las diferentes clases de almacenaje.

En la ausencia de inicializaci¢n expl¡cita, las variables estticas y externas son garantizadas al ser inicializadas por cero; las variables automticas y register tienen valores indefinidos (i.e. basura).

Las variables simples (no-arreglos o estructuras) pueden ser inicializadas cuando ellas son declaradas, siguiendo el nombre con el signo = y una expresi¢n constante :

```
int x = 1;
char squote = ' ";
long day = 60 * 24;    /* minutos en el d¡a  */
```

Para variables externas y estticas, la inicializaci¢n es hecha una vez, conceptualmente a tiempo compilado. Para variables automticas y register esto es hecho cada vez que la funci¢n o bloque es entrada.

Para variables automticas y register, el inicializador no est restringido a ser una constante : puede, en efecto, ser una expresi¢n involucrando valores definidos previamente, a£n llamados de funci¢n. Por ejemplo, las inicializaciones del programa de b£squeda binaria en el cap.3 puede ser escrito como

```
binary(x,v,n)

int x,v  ,n;

    int low = 0;
    int high = n-1;
    int mid;
        ...
```

en lugar de

```
binary(x,v,n)

int x, v  ,n;


    int low, high, mid;

        low = 0;
        high = n-1;
            ...
```

En efecto, las inicializaciones de variables automticas son justamente una taquigraf¡a para asignamientos de instrucciones. cuya forma de preferencia es cuesti¢n de gusto. Generalmente hemos usado asignamientos expl¡citos, porque los inicializadores en declaraciones son dif¡ciles de ver.

Los arreglos automticos no pueden ser inicializados siguiendo la declaraci¢n con una lista de inicializadores encerrados entre corchetes y separados por comas. Por ejemplo, el programa que cuenta caracteres del cap.1, el que comenzar

```
main ()   /* cuenta d¡gitos, espacios en blanco y otros
                        */
    int c, i, nwhite, nother;
    int ndigit 10 ;

    nwhite = nother = 0;
    for(i = 0; i < 10; i++)
        ndigit i  = 0;
        ...
```

puede ser escrito como

```
int nwhite = 0;
int nother = 0;
int ndigit 10  =  0, 0, 0, 0, 0, 0, 0, 0, 0, 0  ;


main()   /* cuenta d¡gitos, espacios en blancos y otros
                        */

    int c, i;
     ...
```

Estas inicializaciones son actualmente innecesarias ya que todos son ceros, pero es una buena forma para hacerlos expl¡citos sea lo que fuere. Si hay menos inicializadores que el largo especificado, los otros sern ceros. Es un error tener almacenados inicializadores. Lamentablemente, no hay forma para especificar la repeticie¢n de un inicializador, no para inicializar un elemento en la mitad del arreglo sin suplir todos los valores intervinientes.

Los arreglos de caracteres son un caso especial de inicializaci¢n; un string puede ser usado en lugar de los corchetes y comas

```
char pattern   = "the";
```

Esta es una taquigraf¡a para el ms largo pero equivalente a :

```
char pattern   =  't', 'h', 'e', ' 0' ;
```

Cuando el largo de un arreglo de alg£n tipo es omitido, el compilador calcula el largo contando los inicializadores. En este caso espec¡fico, el largo es 4 (tres caracteres mas el t,rmino 0).

4.10 RECURSION

Las funciones de C pueden ser usadas recursivamente; esto es, una funci¢n puede llamar por si misma directa o indirectamente. Un ejemplo
tradicional imprime un n£mero como un string de caracteres. Como mencionamos antes, los d¡gitos son generados en el orden equivocado : los d¡gitos de menos orden son compatibles antes que los d¡gitos de alto orden, pero ellos tienen que ser impresos en la otra forma.

Hay dos soluciones para este problema. Una es almacenar los

d¡gitos en un arreglo como ellos estn generados, entonces los imprimen en el orden inverso, como lo hicimos en cap.3 con itoa. La primera versi¢n de printd sigue este modelo.

```
printd(n)   /* imprime n en decimal  */
int n;

    char s 10 ;
    int i;
    if(n < 0)
        putchar('-');
        n = -n;

    i = 0;
    do
        s i++  = n % 10 + '0';   /* encuentra pr¢ximo
                          carácter  */
      while((n /= 10) > 0);   /* descartarlo  */
    while(--i >= 0)
        putchar(s i );
```

La alternativa es una soluci¢n recursiva, en que cada llamada de printd primero llama lo mismo para confrontar con algunos d¡gitos principales, entonces imprime el d¡gito arrastrado.

```
printd(n)      /* imprime n en decimal (recursivo)  */
int n;

    int i;

    if(n < 0)
        putchar('-');
        n = -n;
    if((i = n/10) != 0)
        printd(i);
    putchar(n % 10 + '0');
```

Cuando una funci¢n llama lo mismo recursivamente, cada invocaci¢n entrega un nuevo set de todas las variables automticas, completamente independiente del set previo. As¡ en printd(123) el primer printd tiene n = 123. Pasa 12 a un segundo printd entonces imprime 3 cuando ese retorna. En la misma forma, el segundo printd pasa 1 a un tercero (el que lo imprime), entonces imprime 2. La recursi¢n generalmente no proporciona un grabado en el almacen, en vista de que en alguna parte los valores de un stack estn siendo procesados para ser guardados. No ser rpido. Pero un c}digo recursivo es ms compacto, y frecuentemente m#s fcil de escribir y entender. La recursi¢n es especialmente conveniente para estructura de datos definidos recursivamente, semejantes a rboles; veremos un elegante ejemplo en el cap.6.

4.11 EL PROCESADOR C

C proporciona ciertas extenciones del lenguaje para conocer el significado de un simple procesador macro. La capacidad del define que hemos uzado es el ms com£n de estas extenciones; otra es la habilidad para incluir el contenido de otros archivos durante la compilaci¢n.

INCLUSION DE ARCHIVOS

Para facilitar el manejo de colecciones de define's y (declaraciones entre otras cosas) C proporciona un archivo de inclusi¢n. Alguna línea que sea semejante a

```
include "nombre-archivo"
```

es reemplazada por el contenido del archivo nombre-archivo. (Las comillas son obligatorias.) Frecuentemente una línea o dos de esta forma aparecen al comienzo de cada archivo fuente, para incluir instrucciones define y declaraciones extern para variables globales. Los include's pueden anidados.

El include es la manera preferida para enlazar las declaraciones juntas para un programa largo. Garantiza que todos los archivos fuente sern abastecidos con las mismas definiciones y declaraciones de variables, y as¡ eliminar una clase de error particularmente intrable. Cuando un archivo inclu¡do es cambiado, todos los archivos que dependen
de el deben ser recompilados.

SUSTITUCION MACRO

Una definici¢n de la forma

```
define  YES  1
```

llama una sustituci¢n macro de la clase ms simple - reemplazando un nombre por un string de caracteres. Los nombres en define tienen la misma forma como los identificadores C; el reemplazo del texto es arbitrario. Normalmente el reemplazo de texto es el resto de la línea; una gran definici¢n puede ser continuada ubicando un al final de la línea para ser continuada. El "alcance" de un nombre definido con define es desde su punto de definici¢n al final del archivo fuente. Los nombres pueden ser redefinidos, y una definici¢n puede usar definiciones previas. Las sustituciones que no tienen lugar en el string entre comillas, así por ejemplo, si YES es un nombre definido, no habr¡a sustituci¢n en printf("YES").

Ya que la implementaci¢n de define es una macro prepass, no parte del propio compilador, hay muy pocas restricciones gramaticales en que puede ser definida. Por ejemplo, Algol puede decir

```
define  then
define begin
define end    ;
```

y entonces escribir

```
if (i > 0) then
    begin
        a = 1;
        b = 2
    end
```

Tambi¸n es posible definir macros con argumentos, as¡ el reemplazo de texto depende de la forma en que la macro es llamada. Como un ejemplo, definamos la macro llamada max, semejante a esto :

```
define max(A, B) ((A) > (B) ? (A) : (B))
```

ahora la línea

```
x = max(p + q, r + s);
```

ser reemplazada por la línea

```
x = ((p + q) > (r + s) ? (p + q) : (r + s));
```

Esto proporciona una funci¢n "maximo" que despliega el c¢digo en línea mejor que una llamada de funci¢n. Mientras que los argumentos
son tratados consistentemente, esta macro servir para algn tipo de datos, como habr¡a con las funciones.

Si Ud. examina la expansi¢n del max anterior notar¡a algunas trampas. Las expresiones son evaluadas dos veces; esto es malo si ellas involucran efectos laterales como llamados de funci¢n y operadores
de incremento. Alg£n cuidado ha de ser tomado con los par¸ntesis para

asegurar que el orden de evaluaci¢n es preservado. (Considerar la macro

```
define square(x)  x * x
```

que involucra la square(z+1).) Hay, a£n, algunos problemas puramente l,xicos : all¡ no puede estar el espacio entre el nombre de la macro y el par,ntesis izquierdo que lo introduce a la lista de argumentos.

Ninguna de las macros son absolutamente valiosas. Un ejemplo prctico es la biblioteca standard de I/O que ser descrita en el cap.7, en que getchar y putchar son definidas como macros (obviamente putchar
necesita un argumento), as¡ evitando el encabezamiento de una funci¢n llamada por carácter procesado.

Otras capacidades del procesador macro son descritas en el ap,ndice A.

_

CAPITULO 5 PUNTEROS Y ARREGLOS

Por Rolando ("GALAN DEL CIISA") CORIMA

Un puntero es una variable que contiene las dirreciones de otra variable. Los punteros son muy usados en C, en parte porque ellos son, a veces, la nica manera de expresar un computo, y en parte porque ellos usualmente están dirigidos para códigos mas compactos y eficientes que pueden ser obtenidos en otras formas.

los punteros han sido amontonados con la instrucción goto como una maravillosa forma para crear programas "imposibles de entender". Esto es cierto cuando ellos son usados cuidadosamente, y esto es facil para crear punteros que apunten a alguna parte inesperada. Con disciplina, los punteros pueden también ser usados para codeguir claridad y simplicidad. Este es el aspecto que prataremos de ilustrar.

5.1 PUNTEROS Y DIRECCIONES

Desde que un puntero contiene la direccion de un objeto, es posible accesar el objeto "indirectamente" a traves del puntero. Supongamos que x es una variable, un int, y que px es un puntero, creado en alguna forma sin especificar el operador unario & entrega la direccion de un objeto, así, las instrucciones

```
px = &x;
```

asigna la direccion de x a la variable px. El operador & puede ser aplicado solo a las variables y elementos de arreglos; construcciones semejantes a &(x+1) y &3 son ilegales. Es también ilegal tomar la direccion de un "registro" variable.

El operador unario * trata su operador como la direcci¢n de la ultima tarjeta, y accesar esa direccion para ir a buscar el contenido. Así, si y también es un int,

```
y = *px;
```

asigna a y el contenido de todo lo que px aounte. Así la secuencia

```
px = &x;
y  = *px;
```

asigna el mismo valor para y como hace

```
y = x;
```

Si es necesario declarar las variables que participan en todo esto :

```
int x, y;
int *px;
```

La declaración de x e y es tal como lo hemos visto anteriormente. La declaración del puntero es nueva.

```
int +px;
```

es entendida como una nemotecnica; dice que la combinacion *p es un entero, esto es, si px ocurre en el contexto *p, es equivalente a una variable del tipo int. En efecto, la sintaxis de la declaración para una variable mimica la sintaxis de expresiones en que la variable puede aparecer. Este razonamiento es til en todos los casos que involucran declaraciones complicadas. Por ejemplo,

```
double atof(), *dp;
```

dice que en una expresión atof(), y *dp tienen valores del tipo double.

Ud. debería también notar la implicacion en la declaración, que un puntero esta restringido a un punto de un genero particular de objeto.

Los punteros pueden ocurrir en expresiones. Por ejemplo, si px apunta al entero x, entonces *px puede ocurrir en algn contexto donde x podria ocurrir.

```
y = *px + 1
```

coloca en y uno mas que x;

```
printf ("%d n", *px)
```

imprime el valor en curso de x y

```
d = sqrt ((double) *px)
```

produce en d la raiz cuadrada de x, que es restringida a un double antes de que esten siendo pasados a sqrt (ver cp.2)

En expresiones como :

```
y = *px + 1
```

los operadores unarios * y & ligan mas apretadamente que los operadores
aritmeticos, así estas expresiones toman todo lo que apunte px, suma 1, y lo asigna a y. Volveremos dentro de poco para ver que significado podria tener :

```
y = *(px + 1)
```

Las referencias de punteros también pueden ocurrir en el lado izquierdo de los asignamientos. Si px apunta a x, entonces

```
*px = 0
```

pone un cero en x, y

```
*px += 1
```

lo incrementa, así como

```
(*px)++
```

Los paréntesis son necesarios en este ultimo ejemplo; sin ellos, la expresión incrementaria px en vez de que lo apunte, porque los operadores unarios como * y ++ son evaluados de derecha a izquierda.

Finalmente, ya que los punteros son variables, ellos pueden ser manipulados como otras variables. Si py es otro puntero para int, entonces

```
py = px
```

copia el contenido de px en py, así haciendo que py apunte a todo lo que px apunte.

5.2 PUNTEROS Y ARGUMENTOS DE FUNCION

Ya que C pasa los argumentos a las funciones por "llamada por valores", no hay una forma directa de alterar una variable para la función llamada.] Que hace Ud. si realmente tiene que cambiar un argumento comn ? Por ejemplo, una rutina que sortea puede cambiar 2 elementos fuera de orden con una función llamada swap. No es bastante
para escribir

```
swap(a, b);
```

donde la función swap esta definida como

```
swap(x, y)      /*WRONG*/
int x, y;

    int temp;

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

Porque de la llamada por valor, swap no puede afectar los argumentos a y
b en la rutina que lo llamo.

Afortunadamente, hay una manera para obtener el efecto deseado. El programa de llamada pasa los punteros a los valores para ser clasificados :

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

ya que el operador & entrega la direccion de una variable, &a es un puntero para a. En swap mismo, los argumentos son declarados para ser
punteros, y los operandos actuales son accesados a traves de ellos.

```
swap(px, py)   /* intercambia *px y *py */
int *px, *py;

    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
```

Un uso comn de argumentos de punteros es en funciones que deben devolver mas de un simple valor. (Ud. podria decir que swap devuelve dos valores, el nuevo valor de sus argumentos.) Como un ejemplo, considerar una función getint que ejecute un formato libre de conversión de entrada para interrumpir una corriente de caracteres en valores enteros, un entero por llamada. getint ha de devolver al valor encontrado, o una se|al de fin de archivo cuando no hay mas entrada. Estos valores tienen que ser devueltos como objetos separados, pues no importa que valor es usado por EOF que podria ser también al valor de un
entero de entrada.

Una solucion, que esta basada en la función de entrada scanf que describiremos en el **CAPITULO** 7, es tener getint que devuelve EOF como su
valor de función si lo encontro al final del archivo; alguna otra devolvera un "valor senal" de un entero normal. Esta organizacion separa el fin de archivo desde los valores numericos.

El siguiente loop llena un arreglo con enteros llamados por getint

```
int n, v, array  size  ;
for (n = 0; n < size && getint(&v) != EOF; n++)
    array n  = v;
```

Cada llamada pone en pbvpb el próximo entero encontrado en la entrada. Notar que esto es esencial para escribir &v en vez de v como el argumento de getint. Usando v es probable causar un error de direccionamiento, desde que getint lo cree ha sido dirigido a un puntero valido.

getint mismo en una modificacion obvia de atoi que escribimos antes:

```
getint(pn)    /* entrega el próximo entero desde la
                 entrada */
int *pn;

    int c, sign;

    while ((c = getch()) == ' ' !! c == ' n' !!
                c == ' t')
        ;       /* salta espacio en blanco  */
    sign = 1;
    if (c == '+' !! c == '-')    /* record sign */
        sign = (c == '+') ? 1 : -1;
    c = getch();

for (*pn = 0; c >= '0' && c <= '9'; c = getch())
    *pn = 10 * *pn + c - '0';
```

```
    *pn *= sign;
    if (c != EOF)
        ungetch(c);
    return(c);
```

Por todo getint, *pn es usado como una variable intPb ordinaria. Hemos
también usado getch y ungetch (descrito en el cap.4) así el carácter extra que debe ser leído puede ser devuelto a la entrada.

5.3 PUNTEROS Y ARREGLOS

En C, hay una fuerte relacion entre los punteros y arreglos, de manera que punteros y arreglos deberan ser tratados simultaneamente. ALguna operación que puede ser lograda por subscripcion de arreglo, puede ser también hecha con punteros. La versión del puntero en general será rapida pero, pero por lo menos para el no-iniciado, algo dificil de asimilar rápidamente.

La declaración

```
    int a 10
```

define un arreglo de largo 10, esto es un block de 10 objetos consecutivos llamados a o , a 1 ,...,a 9 . La notación a i quiere decir que el elemento i del arreglo se poseciona desde el comienzo. Si pa es un puntero para un entero, declarado como

```
    int *pa
```

entonces el asignamiento

```
    pa = &a 0
```

coloca pa apuntando el elemnto cero de a; este es, pa contiene la direccion de a 0 . Ahora el asignamiento

```
    x = *pa
```

copiara el contenido de a 0 en x.

si pa apunta a un elemento particular de un arreglo a, entones por definición pa+1 apunta al próximo elemento, y en general pa-i apunta i elementos antes de pa, y pa+i apunta i elementos después. Así, si pa apunta a a 0 ,

```
    *(pa+1)
```

se refiere al contenido de a 1 , pa+1 es la direccion de a i , y *(pa+i) es el contenido de a i .

Estas observaciones son verdaderas, descuidado el tipo de las variables en el arreglo a. La definición de " sumando 1 al puntero", y por extension, todo puntero aritmético, es que el incremento esta escalado por el largo en almacenaje del objeto que es apuntado. As¡ en pa+i, i es multiplicado por el largo de los objetos que pa apunta de que sea sumado a pa.

La correspondencia entre indexar y un puntero aritmético es evidentemente muy cerrada. En efecto, una referencia para un arreglo es convertido por el compilador a un puntero al comienzo del arreglo. El efecto es que un nombre de arreglo es una expresi¢n puntero. Esto tiene unas pocas implicaciones £tiles.ya que el nombre de un arreglo es un sin¢nimo para la localizaci¢n del elemento cero, el asignamiento

```
    pa = &a 0
```

tambi,n puede ser escrito como

```
    pa = a
```

Mas sorpresa, por lo menos a primera vista, es el hecho que una referencia a a i tambi,n puede ser escrita como *(a+i). Evaluando a i , C lo convierte a *(a+i) inmediatamente; las dos formas son completamente equivalentes. Aplicando el operador & a ambas partes de
esta equivalencia, lo sigue ese &a i y a+i son tambi,n id,nticos: a+i es la direcci¢n del i-,simo elemento despu,s de a. Como el otro lado de esta moneda, si pa es un puntero, las expresiones pueden usarlo con un sub¡ndice: pa i es id,ntico a *(pa+i). En corto, alg£n arreglo e ¡ndice de expresi¢n pueden ser escritos como un puntero y offset, y viceversa, a£n en la misma instrucci¢n.

Hay una diferencia entre un nombre de arreglo y un puntero que debe tenerse presente. Un puntero es una variable, as¡ pa=a y pa++ son operaciones con sentido. Pero un nombre de arreglo es una constante, no una variable: construcciones como a=pa o a++ o p=&a son ilegales.

Cuando un nombre de arreglo es pasado a una funci¢n, lo que pasa es la ubicaci¢n del comienzo del arreglo. Dentro de la funci¢n llamada, este argumento es una variable, justo como alguna otra variable, y as¡ un argumento de nombre de arreglo es verdaderamente un puntero, esto es,
una variable conteniendo una direcci¢n. Podemos usar este hecho para escribir una nueva versi¢n de strlen, que calcula el largo de un string.

```
    strlen(s)     /* devuelve el largo del string s  */
    char *s;

        int n;

        for (n = 0; *s != ' 0'; s++)
            n++;
        return(n);
```

Incrementar s es perfectamente legal, ya que es una variable puntero; s++ no tiene efecto sobre el carácter de string en la funnci¢n que llama strlen, pero strlen incrementa una copia privada de la direcci¢n.

Como parmetros formales en una definici¢n de funci¢n,

```
    char s  ;
```

y

```
    char *s;
```

son exactamente equivalentes; que deber¡a ser escrito est determinado connsiderablemente por cuantas expresiones sern escritas en la funci¢n. Cuando un nombre de arreglo es pasado a una funci¢n, la funci¢n puede
creer a su conveniencia que ha sido dirigida por un arreglo o un puntero, y manipularla de consiguiente. Hasta puede usar ambas clases de operaciones si lo ve apropiado y claro.

Es posible pasar parte de un arreglo a una funci¢n, paasando un puntero al comienzo del sub-arreglo. Por ejemplo, si a es un arreglo,

```
    f(&a 2 )
```

y

```
    f(a+2)
```

ambas pasan a la funci¢n f la direcci¢n del elemento a 2 , porque &a 2 y a+2 son ambas expresiones punteros que se refieren al tercer elememto
de a. Dentro de f la declaraci¢n de argumento puede leer

```
    f(arr)
    int arr  ;

        ...
```

o

```
f(arr)
int *arr;

    ...
```

As¡ hasta f est ocupado, el hecho de que el argumento realmente se refiera a la parte de un gran arreglo, no es de consecuencia.

5.4 DIRECION ARITMETICA

Si p es un puntero, entonces p++ incrementa p apuntando al pr¢ximo elemento (un objeto de cualqier clase), y p += i incrementa p apuntando al alemento i despu,s de donde actualmente lo hace. Estas y construcciones similares son la ms simple y com£n forma de puntero o direcci¢n aritm,tica.

C es consistente y regular en su aproximaci¢n a las direcciones aritm,ticas; su integraci¢n de punteros, arreglo y direcci¢n aritm,tica es una de las mayores fuerzas del lenguaje. Ilustraremos algunas de sus propiedades escribiendo un rudimentario "asignador" de almacenaje (pero £til a pesar de su simplicidad). Hay dos rutinas: alloc(n) devuelve un puntero p a n posiciones consecutivas de caracteres, que pueden ser usadas por el que llama a alloc para almacenar caracteres; free(p) realiza el almacenaje alcanzado de este modo, as¡ mas tarde puede ser re-usado. Las rutinas son rudimentarias porque los llamados para free deben ser hechos en el orden opuesto a las llamadas hechas en alloc. Esto es, el almacenaje manejado por alloc y free es un stack, o una lista que "sale por el primero" y "entra por el £ltimo". La biblioteca standard de C proporciona funciones anlogas que no tienen tales restricciones, y en el cap.8 moostraremos versiones mejoradas. Mientras tanto, muchas aplicaciones s¢lo necesitan una trivial alloc para distribuir pedacitos de largo impredecible a veces impredecible.

La implementaci¢n ms simple es tener alloc para manipular fuera, pedazos de un gran arreglo de caracteres que llamaremos allocbuf. Este arreglo es privado para alloc y free. Ya que ellos tratan con punteros, no ¡ndices de arreglo, no necesita otra rutina para conocer el nombre del arreglo, y puede ser declarado externo esttico, esto es local para el archivo fuente que contiene alloc y free, y fuera de ,l, invisible. En implementaciones prcticas, el arreglo podr no tener un nombre; en cambio obteni,ndose pidiendo al sistema operativo para alg£n
block de almacenaje que no tenga nombre.

La otra informaci¢n necesitada es cuanto ha sido usado de allocbuf. Usamos un puntero para el pr¢ximo elemento libre, llamado allocp. Cuando alloc es pedido para n caracteres, lo revisa para ver si hay bastante sitio a la izqierda en allocbuf. Si as¡, alloc devuelve el valor actual de allocp (i.e. el comienzo del bloque libre), entonces lo incrementa en n apuntando a la pr¢xima rea libre. free(p) coloca allocp en p si p est dentro de allocbuf.

```
define NULL    0   /* valor del puntero para reportar
                      error */
define ALLOCSIZE  1000  /* largo de espacio
            aprovechable  */
static char  allocbuf ALLOCSIZE ;   /*almacenaje para
                           alloc  */
static char *allocp = allocbuf;  /* pr¢xima posici¢n
                        libre  */

char *alloc(n) /* devuelve puntero para n caracteres
                      */
int n;

    if(allocp + n <= allocbuf + ALLOCSIZE)   /* ajuste
                                 */
        allocp += n;
        return(allocp - n);   /* antiguo p */
    else          /* no hay bastante espacio  */
        return(NULL);
free(p)       /* almacenaje libre apuntado por p  */
char *p;

    if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
        allocp = p;
```

Algunas declaraciones. En general un puntero puede ser inicializado justo como alguna otra variable, aunque normalmente el £nico significado es el valor NULL (discutido anteriormente) o una expresi¢n que involucra direcciones de datos de tipo apropiado, definidos previamente. La declaraci¢n

```
static char *allocp = allocbuf;
```

define allocp para ser un carácter puntero y lo inicializa para apuntar a allocbuf, que es la pr¢xima posici¢n libre donde parte el programa. Esto tambi,n podr¡a ser escrito como

```
static char *allocp = &allocbuf 0 ;
```

ya que el nombre del arreglo es la direcci¢n del elemento cero; cualquiera que se use es sencillo.

La pregunta

```
if(allocp + n <= allocbuf + ALLOCSIZE)
```

revisa si hay bastante espaciopara satifacer un requerimiento para n caracteres. Si hay, el nuevo valor de allocp estar a lo ms uno despu's dl fin de allocbuf. si l requerimiento puede ser satisfecho, alloc devuelve un puntero normal (notar la declaraci{n de la funci¢n misma). Si no, alloc debe devolver alguna se|al de que no hay espacio a la izquierda. C garantiza ese no-puntero que vlidamente apunta el dato que contiene al cero, as¡ un valor devuelto de cero puede ser usado para se|alar un evento anormal, en este caso, no-espacio. Escribimos NULL en vez de cero, sin embargo, para indicar mas claramente que hay un
valor especial para un puntero. En general, los enteros no pueden ser integramente asignados a punteros; cero es valor especial.

Preguntas como

```
if (allocp + n <= allocbuf + ALLOCSIZE)
```

y

```
if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
```

muestran varias facetas importantes de puntero aritm,tico. Primero, los punteros deben ser comparados bajo ciertas circunstancias. Si p y q apuntan a miembros del mismo arreglo, entonces relaciones como <, >=,
etc., trabajan apropiadamente.

```
p < q
```

es verdadero, por ejemplo, si p apunta a un miembro del arreglo antes de que lo haga q. Las relaciones == y != tambi,n trabajan. alg£n puntero puede ser completamente comparado por igualdad o desigualdad con
NULL. Pero todas las apuestas estn fuera si Ud. hace aritm,tica o comparaciones con punteros apuntando a diferentes arreglos. Si Ud. es afortunado, obtendr un disparate en todas las mquinas. Si Ud. no tiene suerte, su c¢digo trabajar en una mquina, pero se derrumbar misteriosamente en otra.

Segundo, ya hemos observado que un puntero y un entero pueden ser sumados o restados. La construcci¢n

```
p + n
```

significa que el n-,simo objeto et apuntado por el p en curso. esto es verdad descuidando el g,nero del objeto, p es declarado para apuntar a; el compilador grad£a n de acuerdo al largo de los objetos que apunta p, que est determinado por la declaraci¢n de p. Por ejemplo, en el PDP-11, las escalas de factores son 1 para char, 2 para int y chort, 4 para long y float, y 8 para double.

La substracci¢n de punteros es tambi,n vlida: si p y q apuntan a miembros del mismo arreglo, p-q es el n£mero de elemntos entre p y q. Este hecho puede ser usado paara escribir otra versi¢n de strlen:

```
strlen(s) /* devuelve largo de string s */
char *s;

    char *p = s;

    while (*p != ' 0')
        p++;
    return(p-s);
```

En la declaraci¢n, p es inicializado en s, esto es, apunta al primer caracter. En el while, cada caracter es examinado hasta que 0 es visto al final. Ya que 0 es cero, es posible omitir la pregunta expl¡cita, y cada loop es escrito frecuentemente como

```
while(*p)
    p++;
```

Porque p apunta a caracteres, p++ avanza p al pr¢ximo caracter cada vez, y p-s entrega el n£mero de caracteres avanzados, esto es, el largo del string. El puntero aritm,tico es consistente; si hemos estado procediendo con float's, que ocupan mas almacenaje que los char's, meramente cambiando char a float por todo alloc y free. Todas las manipulaciones automticamente toman los punteros considerando el largo
del objeto apuntado, as¡ ning£n otro ha de ser alterado.

Otra de las operaciones mencionadas aqu¡ (sumando o restando un puntero y un entero; restando o comparando dos punteros), todo otro puntero aritm,tico es ilegal. No est permitido, multiplicar, dividir, desviar u ocultarlos, o sumar float o double a ellos.

5.5 PUNTEROS DE CARACTERES Y FUNCIONES

Un string constante, escrito como

```
"yo soy un string"
```

es un arreglo de caracteres. En la representaci¢n interna, el compilador termina el arreglo con el caracter 0 as¡ ese programa puede encontrar el final. El largo en almacenaje es as¡, uno ms que el n£mero de caracteres entre comillas.

Quizs la ocurrencia ms com£n de string constante es como argumento para funci¢n, como en

```
printf ("Hola, mundo n");
```

Cuando un string de caracter como este aparece en un programa, lo accesa
a trav,s de un puntero caracter; lo que printf recibe es un puntero para el arreglo de caracter.

Los arreglos de caracter en curso no necesitan ser argumentos de funci¢n. Si message es declarado como

```
char *message
```

entonces la instrucci¢n

```
message = "ahora es tiempo";
```

asigna a message un puntero para los caracteres actuales. Esto no es una copia del string; s¢lo involucra punteros. C no proporciona algunos operadores, para procesar un string completo con caracteres, como una unidad.

Ilustraremos ms aspectos de punteros y arreglos estudiando dos funciones £tiles desde la biblioteca standard de I/O que discutiremos en el cap.7.

La primera funci¢n es strcpy(s, t), que copia el string t al string s. Los argumentos son escritos en este orden por analog¡a al asignamiento, donde uno podr¡a decir

```
s = t
```

para asignar t a s. La primera versi¢n del arreglo es:

```
strcpy(s, t)       /* copia t a s  */
char s  , t  ;

    int i;

    i = 0;
    while ((s i  = t i  ) != ' 0)
        i++;
```

Por el contrario, hay una versi¢n de strcpy con punteros.

```
strcpy(s, t)       /* copia t a s; versi¢n 1, puntero
                        */
char *s, *t;

    while ((*s = *t) != ' 0'
        s++;
        t++;
```

Porque los argumentos son pasados por valor, strcpy puede usar s y t en la forma que guste. Aqu¡ los punteros son convenientemente inicializados, que son puestos en marcha a lo largo de los arreglos un caracter a la vez, hasta que el 0 que termina t ha sido copiado a s.

En la prctica, strcpy no deber¡a ser escrito como mostramos arriba. Una segunda posibilidad puede ser

```
strcpy(s, t)       /* copia t a s; versi¢n 2, puntero
                        */
char *s, *t;

    while ((*s++ = *t++) != ' 0')
        ;
```

Esto mueve el incremento de s y t dentro de la parte de la pregunta. El valor de *t++ es el caracter que apunta t antes de que t fuera incrementado; el sufijo ++ no cambia t hasta despu,s de que este caracter ha sido tra¡do. En la misma forma, el caracter es almacenado en la antigua posici¢n s antes que s sea incrementado. Este caracter es tambi,n el valor que es comparado contra 0 para controlar el loop. El efecto neto es que los caracteres son copiados desde t a s incluyendo la terminaci¢n 0.

Como la abreviaci¢n final, observemos nuevamente que una comparaci¢n contra 0 es redundante, as¡ la funci¢n es frecuentemente escrita como

```
strcpy(s, t)       /*copia t a s; versi¢n 3, puntero
                        */
char *s, *t;
```

```
while(*s++ = *t++)
    ;
```

Aunque esto puede parecer escondido a primera vista, la conveniencia notacinal es considerable, y el idioma deber¡a ser dominado, si por ninguna otra raz¢n que esa, Ud. lo ver frecuentemente en programas C.

La segunda rutina es strcmp(s, t), que compara los strings de caracteres s y t, y devuelve negativo, cero o positivo de acuerdo a si s es lexicogrficamente menor que, igual a, o mayor que t. El valor devuelto es obtenido restando los caracteres a la primera posici¢n donde s y t no son convenientes.

```
strcmp(s,t)        /* devuelve <0 si s<t, 0 si s==t,
                  >0 si s>t  */
char s  , t  ;

    int i;

    i = 0;
    while (s i  == t i )
        if ( s i++  == ' 0')
                return(0);
    return(s i   - t i  );
```

La versi¢n puntero de strmcp:

```
strcmp(s, t)
char  *s, *t;

    for ( ; *s == *t; s++, t++)
        if (*s == ' 0')
            retun(0);
    retun(*s - *t);
```

Ya que ++ y -- son operadores sufijos o prefijos, otras combinaciones de *, ++ y -- ocurren, aunque menos frecuentemente. Por ejemplo,

```
*++p
```

incrementa p antes de ir a buscar el caracter que p apunta;

```
*--p
```

primero decrementa p.

5.6 PUNTEROS NO SON ENTEROS

Ud. puede notar en viejos programas C, una mejor actitud con respecto a copiar punteros. Generalmente, en muchas mquinas un puntero puede ser asignado a un entero y devolverlo nuevamente sin cambiarlo; lamentablemente, esto ha conducido a la toma de libertades con rutinas que devuelven punteros que son pasados meramente a otra rutina - las declaraciones de punteros frecuentemente estn afuera. Por ejemplo, considerar la funci¢n strsave(s), que copia el string s en un lugar seguro, obtenido por una llamada en alloc, y devuelve un puntero a este. Propiamente, esto deber¡a ser escrito como

```
char *strsave(s)    /* graba string s en alguna parte
                       */
char *s;

    char *p, *alloc();

    if ((p == alloc(strlen(s) + 1)) != NULL)
        strcpy(p,s);
    retun(p);
```

En la prctica, all¡ deber£a estar una fuerte tendencia para omitir declaraciones:

```
strsave(s)        /*graba string s en alguna parte
                    */

    char *p;

    if((p = alloc(strlen(s) + 1)) != NULL)
        strcpy(p, s);
    return(p);
```

Este trabajar en muchas mquinas, ya que el tipo omitido para funciones y argumentos es int, un int y un puntero pueden ser usualmente
asignados atrs y adelante. Nunca esta clase de c¢digo es inherentemente peligroso, pro esto depende en detalle de la implementaci¢n y arquitectura de la mquina que puede no tener cabida para el compilador particular que Ud. use. Es sensato completar en todas las declaraciones. (El programa lint avisar de tales construcciones, en caso que ellos se deslicen dentro, inadvertidamente.)

5.7 ARREGLOS MULTIDIMENSIONALES

C proporciona para rectangular arreglos multidimensionales, aunque en la prctica ellos tienden a ser mucho menos usados que arreglos de punteros. En esta secci¢n, mostraremos algunas de sus propiedades.

Considerar el programa de la conversi¢n de datos, desde d¡a del mes a d¡a del a|o y viceversa. Por ejemplo, 1 de marzo es el 60,simo d¡a de un a|o no bisiesto 61er d¡a de un a|o bisiesto. Definamos dos funciones para hacer las combinaciones: day_of_year convierte el mes y d¡a en el mes del a|o, y month_day convierte el d¡a del a|o en el mes y d¡a. Ya que esta funci¢n ms reciente devuelve dos valores, los argumentos de mes y d¡a sern punteros:

```
month_of_year(1977, 60, &m, &d)
```

coloca en m un 3 y en d un 1 (primero de marzo).

Estas funciones necesitan ambas la misma informaci¢n, una tabla del n£mero de dias en cada mes("30 dias tiene septiembre...). Ya que el n£mero de dias por mes difiere para a|os bisiestos y no bisiestos, es ms fcil separarlas en dos listas de un arreglo bidimensional que intentar guardar el rastro de cuanto le sucede a febrero durante el clculo. El arreglo y las funciones para ejecutar las transformaciones son como sigue:

```
static int day_tab 2  13  =
    0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
  ;

day_of_year(year, month, day)      /* conjunto de dias
                desde mes y dia   */
int year, month, day;

    int i, leap;

    leap = year%4 == 0  && year%100 != 0
            !! year%400 == 0;
    for(i = 1; i < month; i++)
        day += day_tab leap  i ;
    return(day);
```

```
month_day(year, yearday, pmonth, pday)  /* set mes, dia

        */
int year, yearday, *pmonth, *pday;    /* desde dia del
                            a|o  */

    int i, leap;

    leap = year%4 == && year%100 != 0
            !! year%400 == 0;
    for (i = 1; yearday > day_tab leap  i ; i++)
        yearday -= day_tab leap  i ;
    *pmonth = i;
    *pday = yearday;
```

El arreglo day_tab tiene que ser externo a day_of_year y a month_day, as¡ ambas pueden ser usadas.

day_tab es nuestro primer arreglo bidimensional. En C, por definici¢n un arreglo bidimensional es realmente un arreglo unidimensional, cada uno de cuyos elementos es un arreglo. Por consiguiente los sub¡ndices son escritos como

```
day_tab i  j
```

mejor que

```
day_tab i, j
```

como en muchos lenguajes. Adems un arreglo bidimensional puede ser tratado en la misma forma como en otros lenguales. Los elementos son almacenados por filas, esto es, el sub¡ndice de mas a la derecha var¡a tan rpido como elementos son accesados en orden de almacenaje.

Un arreglo es inicializado por una lista de inicializadores entre corchetes; cada fila de un arreglo bidimensional es inicializado por una correspondiente sub-lista. Comenzaremos el arreglo day_tab con una columna en cero, as¡ los n£meros de los meses pueden ir del 1 al 12 en vez del 0 al 11. Ya que el espacio no es de nuestro interes, esto es ms fcil que ajustar ¡ndices.

Si un arreglo bidimensional es pasado a una funci¢n, la declaraci¢n de argumento en la funci¢n debe incluir la dimensi¢n de la columna; la dimensi¢n de la fila es irrelevante, desde que ,sta pasada es, como antes, un puntero. En este caso particular, esto es un puntero para objetos que son arreglos de 13 enteros. As¡ si el arreglo day_tab es pasado a una funci¢n f, la declaraci¢n de f deber¡a ser

```
f(day_tab)
int day_tab 2   13 ;

    ...
```

la declaraci¢n de argumento en f podr¡a tambi,n ser

```
int day_tab     13 ;
```

ya que el n£mero de filas es irrelevante, esto podr¡a ser

```
int (*day_tab)  13 ;
```

lo cual dice que el argumento es un puntero para un arreglo de 13 enteros. Los par,ntesis son necesarios ya que los corchetes tienen mayor precedencia que *; sin par,ntesis, la declaraci¢n

```
int *day tab  13 ;
```

es un arreglo de 13 punteros para enteros, como veremos en la pr¢xima secci¢n.

5.8 PUNTEROS ARREGLOS; PUNTEROS A PUNTEROS

ya que los punteros son variables, Ud. pudo esperar que fuera usado para arreglos de punteros. Verdaderamente, este es el caso. Ilustraremos escribiendo un programa que sortear un set de l¡neas de texto en orden alfab,tico; una versi¢n de UNIX el utilitario sort.

Esto est donde el erreglo de punteros enteros. Si las l¡neas a ser sorteadas son almacenadas completamente en un gran arreglo de caracteres (mantenido por alloc, quizs), entonces cada l¡nea puede ser accesada por un puntero para su primer caracter. Los mismos punteros pueden ser almacenados en un arreglo. Dos l¡neas pueden ser comparadas
pasando sus punteros a strcmp. Cuando dos l¡neas fuera de orden han de
ser cambiadas, los punteros en el arreglo de puntero son cambiados, no las l¡neas de texto mismas. Esto elimina los problemas gemelos de complicada administraci¢n de almacenaje y elevado que podr¡a ir moviendo las actuales l¡neas.

El proceso de sorteo involucran tres pasos:

```
leer todas als l¡neas de entrada
sortearlas
imprimirlas en orden
```

Usualmente, es mejor dividir el programa en funciones que igualen esta natural divisi¢n, con la rutina principal controlando las cosas.

Demoremos el sorteo por un momento, y concentremonos en la estructura de datos, la entrada y la salida. La rutina de entrada ha recogido y grabado los caracteres da cada l¡nea, y construye un arreglo de punteros para las l¡neas. Tambi,n tendr¡a que contar el n£mero de l¡neas entradas, ya que esa informaci¢n es necesitada para sortear e imprimir. ya que la funci¢n de entrada puede trabajar s¢lo con un n£mero finito de l¡neas entradas, esto puede devolver alguna cuenta ilegal de l¡nea semejante a -1 si las entradas presentadas son demaciadas. La rutina de salida s¢lo tiene que imprimir las lines en el orden en que ,stas aparecen en el arreglo de punteros.

```
define NULL   0
define LINES  100      /* mximo de l¡eas a ser
                       sorteadas */

main()        /* sorteo de l¡neas entradas */

    char *lineptr LINES ;    /* punteros para la
                        pr¢xima l¡nea */
    int nlines;         /* n£mero de l¡neas leidas en
                           la entrada */
    if ((nlines = readlines(lineptr, LINES)) >= 0)
        sort(lineptr, nlines);
        writelines(lineptr, nlines),

    else
        printf("entrada demaciada grande para sortear
                                n");



 define MAXLEN    1000

readlines(lineptr, maxlines)       /* lee l¡neas de
                           entrada */
char *lineptr  ;         /* para ordenamianto */


    int len, nlines;
    char *p, *alloc(), line MAXLEN ;

    nlines = 0;
```

```
    while ((len = getline(line, MAXLEN)) > 0)
        if(nlines >= maxlines)
            return(-1);
        else if ((p = alloc(len)) == NULL)
            return(-1);
        else
            line len-1  = ' 0';   /* zap newlines */
            strcpy(p, line);
            lineptr nlines++  = p;

    return(nlines);
```

El newline al final da cada l¡nea es borrado as¡ no afectar el orden en que las l¡neas son sorteadas.

```
    writelines(lineptr, nlines)       /* escribe l£neas de
                                 salida */
    char *lineptr  ;
    int nlines;

        int i;
        for (i = 0; i < nlines; i++)
            printf("%s n", lineptr i );
```

Algo nuevo es la declaraci¢n para lineptr:

```
    char *lineptr LINES ;
```

dice que lineptr es un arreglo de LINES elementos, cada elemento de ,ste
es un puntero para un char. Esto es lineptr i es un caracter puntero, y *lineptr i accesa a un caracter.

Ya que lineptr es por si mismo un arreglo que es pasado a writelines, puede ser tratado como un puntero en exactamente la misma manera como nuestros recientes ejemplos, y la funci¢n puede ser escrita como

```
    writelines(lineptr, nlines)   /* escribe salida de
                             l¡neas */
    char *lineptr  ;
    int nlines;

        while (--nlines >= 0)
            printf("5s n", *lineptr++);
```

*lineptr apunta inicialmente a la primera l¡nea; cada incremento lo avanza a la pr¢xima l¡nea mintras nlines es contado hacia abajo.

Con I/O bajo control, podemos proceder a sortear. El sort shell del cap.3 necesita menos cambios: las declaraciones han de ser modificadas, y la operaci¢n de comparaci¢n debe ser movida dentro de una
funci¢n separada. El algor¡tmo bsico es el mismo.

```
    sort(v, n)     /* sortea los strings v 0  ... v n-1  */
    char *v  ;          /* en orden cresiente */
    int n;

        int gap, i, j;
        char *temp;

        for (gap = n/2; gap > 0; gap /= 2)
            for (i = gap; i < n;i++)
                for (j = i-gap; j >= 0; j -= gap)
                    if(strcmp(v j , v j+gap ) <= 0)
                        break;
                    temp = v j ;
                    v j  = v j+gap ;
                    v j+gap  = temp;
```

Desde que alg£n elemento individual de v (alias lineptr) es un caracter puntero, temp tambi,n deber¡a serlo, as¡ uno puede ser copiado en el otro.

Escribimos el programa tan ¡ntegro como fu, posible, asi como obtenerlo trabajando rpidamente. Puede ser rpido, por instancia, para copiar las pr¢ximas l¡neas directamente en un arreglo mantenido por
readlines, ms bien que copiandolas en line y luego a un lugar oculto mantenido por alloc. Pero es sensato para hacer el promer bosquejo algo fcil de entender, y mas tarde cuidar la "eficiencia". La forma para hacer este programa expresivamente rpido, es probablemente no por
una copia innecesaria de las l¡neas entradas. Reemplazando el shell sort por algo mejor, como Quicksort, es ms probable de hacer una diferencia.

En el cap.1 apuntamos que los loops while y for preguntan por la condici¢n terminal "antes" de ejecutar el cuerpo del loop cada vez, ellos ayudan a asegurar que los programas trabajarn a sus l¡mites, en particular sin entrada. Esto se esta aclarando para caminar a trav,s de las funciones del programa de sorteo, revisando que sucede si no hay texto de entrada.

5.9 INICIALIZACION DE ARREGLOS PUNTEROS

Considerar el problema de escribir una funci¢n month_name(n), que devuelve un puntero a un caracter string conteniendo el nombre del n-,simo mes. Esto es una aplicaci¢n ideal para un arreglo interno esttico. month_name contiene un arreglo privado de caracter string, y devuelve un puntero cuando fue llamado. El t¢pico de esta secci¢n es como ese arreglo de nombres es inicializado.

La sintxis es completamente similar a la inicializaciones previas:

```
    char *month_name(n)      /* devuelve nombre del mes n
                                */
    int n;

        static char *name  =
            "mes ilegal",
            "Enero",
            "Febrero",
            "Marzo",
            "Abril",
            "Mayo",
            "Junio",
            "Julio",
            "Agosto",
            "Septiembre",
            "Octubre",
            "Nobiembre",
            "Diciembre"
         ;
        return((n < 1 !! n > 12) ? name 0  : name n );
```

La declaraci¢n de name, que es un arreglo de caracteres punteros, es el mismo que lineptr en el ejemplo del sorteo. El inicializador es simplemente una lista de caracteres strings; cada uno es asignado a la

correspondiente posici¢n en el arreglo. Ms precisamente, los caracteres del i-'simo string son ubicados en alguna otra parte, y un puntero para ellos es almacenado en name i . Ya que el largo del arreglo name no est especificado, el compilador mismo cuenta los inicializadores y los llena en el n£mero correcto.

5.10 PUNTEROS vs. ARREGLOS MULTIDIMENSIONALES

Los recien llegados a C estan confundidos, algunas veces, acerca de las diferencias entre un arreglo bidimensional y un arreglo de punteros, tales como name en el ejemplo anterior. Dadas las declaraciones

```
int a 10   10 ;
int *b 10 ;
```

el uso de a y b puede ser similar, en que a 5 5 y b 5 5 ambas son referencias legales a un simple entero. Pero a es un arreglo verdadero: todo el almacenaje de las 100 celdas ha sido asignado, y el clculo suscrito al rectngulo convencional es hecho para encontrar alg£n elemento dado. Para b, sin embargo, la declaraci¢n s¢lo ubica 10 punteros; cada uno debe apuntar a un arreglo de enteros. Asumiendo que
cada uno apunta a un arreglo de 10 elementos, luego all¡ estarn 100 celdillas de almacen ubicado al aldo, mas als diez celdillas para los punteros. As¡ el arreglo de punteros usa, libremente, ms espacio, y puede requerir un paso de inicializaci¢n expl¡cito. Pero esto tiene dos ventejas: el acceso a un arreglo es hecho indirectamente a trav,s de un puntero ms bien que por una multiplicaci¢n y una adici¢n, y las filas del arreglo pueden ser de diferentes largos. Esto es, cada elemento de b no necesita apuntar a un vector de 10 elementos; alguno puede apuntar a dos elementos, algunos a 20, y algunos a nada.

Aunque hemos fraseado esta discusi¢n en t,rminos de enteros, lejos del uso ms frecuente de arreglos de punteros es semejante al mostrado en month_name: para guardar strings de caracteres de distintos largos.

5.11 ARGUMENTOS DE LINEA DE COMANDO

En el medio ambiente que sostiene C, hay una forma de pasar argumentos de l¡nea de comando o parmetros a un programa donde lo comience ejecutando. Cuando main es llamado para comenzar la ejecuci¢n, es llamado con dos argumentos. El primero (convencionalmente llamado argc) es el n£mero de argumento de l¡nea de
comando con que fu, invocado el programa; el segundo (argv) es un puntero para un arreglo de caracter string que contiene los argumentos, uno por strings. Manipulando estos string de caracter es un uso com£n de multiples niveles de punteros.

La ilustraci¢n y uso de las declaraciones necesarias ms simples es el programa echo, que simplemente devolver sus argumentos de l¡nea de comando en una simple l¡nea, separada por blancos. Esto es, si el comando

```
echo hola, mundo
```

es dado, la l¡nea es

```
hola, mundo
```

Por convenci¢n, argv 0 es el nombre por el que el programa fu, invocado, as¡ argc es al menos 1. En el jemplo anterior, argc es 3, y argv 0 , argv 1 y argv 2 son "echo", "hola" y "mundo" respectivamente. El primer argumento real es argv 1 y el £ltimo es argv argc-1 . Si argc es 1, no hay argumentos de l¡nea de comando despu,s del nombre de
programa. Esto es mostrado en echo:

```
main(argc,argv)     /* argumentos echo; 1ra versi¢n */
int argc;
char *argv  ;
int i;
for (i = 1; i < argc; i++)
    printf("%s %c", argv i , (i < argc - 1) ?
            ' ': ' n');
```

Ya que argv es un puntero para un arreglo de punteros, hay varias maneras de escribir este programa que involucra manipular el puntero mas
bien que indicando un arreglo. mostraremos dos variaciones.

```
main(argc, argv)    /* argumentos echo, 2da. versi¢n
                                */
int argc;
char *argv  ;

    while (--argc > 0)
        printf ("%s %c", *++argv, (argc > 1)
                    ? ' ' : ' n');
```

Ya que argv es un puntero para el comienzo del arreglo de argumento strings, incrementandolo en 1 (++argv) lo hace apuntar al argv 1 original en vez de argv 0 . Cada incremento sucesivo lo mueve a lo largo del pr¢ximo argumento; *argv es entonces el puntero para ese argumento. Al mismo tiempo, argc es decrementado; cuando llega a ser
cero, no hay argumentos a la izquierda para imprimir.

Alternativamente,

```
main(argc, argv)    /* 3ra. versi¢n */
int argc;
char *argv  ;

    while (--argc > 0)
        printf((argc > 1) ? "%s" : "%s n",*++argv);
```

sta versi¢n muestra que el formato de argumento de printf puede ser una expresi¢n justo como alguna de las otras. Este uso no es muy frecuente, pero vale recordarlo.

Como un segindo ejemplo, haremos algunas mejor¡as al programa que
encuentra el modelo, del cap.4. Si Ud. recuerda, instalamos la b£squeda del modelo en el "fondo" del programa, un arreglo obviamente
insatisfactorio. Siguiendo la delantera del utilitario grep de UNIX, cambiamos el programa, as¡ el modelo que ser marcado es especificado por el primer argumento en la l¡nea de comando

```
define MAXLINE   1000
main(argc,argv)     /* encuentra el modelo desde el
                    primer argumento */

int argc;
char *argv  ;

    char line MAXLINE ;
    if (argc != 2)
        printf("uso : encuentra el modelo n");
    else
        while (getline(line, MAXLINE) > 0)
            if (index(line, argv 1 ) >= 0)
                printf("%s", line);
```

El modelo bsico ahora puede ser elaborado para ilustrar mas construcciones de punteros. Suponga que deseamos permitir dos

argumentos opcionales. Uno dice "imprima todas las l¡neas excepto las que contienen el modelo;" el segundo dice "precede cada l¡nea impresa con n£mero de l¡nea".

Una convensi¢n com£n para programas C es que un argumento que comienza con un signo menos introduce un flag opcional o parmetro. S¡
elegimos -x (para "excepto") para se|alar la inversi¢n, y -n ("n£mero") para solicitar numeraci¢n de l¡nea, entonces el comando

```
find -x -n the
```

con la entrada

```
now is the time
for all good men
to come to the aid
of their party
```

deber¡a producir la salida

```
2 : for all good men
```

Los argumentos opcionales deber¡an ser permitidos en alg£n orden, y el resto del programa ser¡a insensible al n£mero de argumentos que actualmente estuvieron presentes. En particular, el llamado para index no se refiere a argv 2 cuando hubo un simple argumento y a argv 1 cuando no hubo. Adems es conveniente para los usuarios si la opci¢n de argumentos pueden ser concatenados, como en

```
find -nx the
```

Aqu¡ est el programa.

```
define MAXLINE      1000

main(argc, argv)    /* encuentra modelo desde el primer
                       argumento */
int argc;
char *argv  ;

    char line MAXLINE , *s;
    long lineno = 0;
    int except = 0, number = 0;

        while (--argc > 0 && (*++argv) 0  == '-')
            for (s = argv 0  + 1; *s != ' 0'; s++)
                switch (*s)
                case 'x':
                    except = 1;
                    break;
                case 'n':
                    number = 1;
                    break;
                default:
                   printf("encuentra opci¢n ilegal
                           %c n", *s);
                    argc = 0;
                    break;

        if (argc != 1)
            printf ("uso: encuentra -x -n modelo
                  n");
        else
            while (getline(line, MAXLINE) > 0)
                lineno++;
                if ((index(line, *argv) >= 0) !=
                      except)
                    if (number)
                        printf ("%1d: ", lineno);
                    printf ("%s", line);
```

argv es decrementado antes de cada argumento opcional, y argc. Si no hay errores, el final del loop argc ser¡a 1 y
*argv apuntar¡a al modelo. Notar que *++argvpb es un puntero para un argumento string; (*++argv) 0 es su primer caracter. Los par,ntesis son necesarios, porque sin ellos la expresi¢n ser¡a *++(argv 0), que es completamente diferente (y equivocada). Una forma alternativa vlida ppdr¡a ser **++argv.

5.12 PUNTEROS PARA FUNCIONES

En C, una funci¢n no es una variable, pero es posible definir un "puntero para una funci¢n", que puede ser manipulado, pasado a funciones
y ubicado en arreglos. Ilustraremos esto modificando el procedimiento de ordenamiento escrito tempranamente en este cap¡tulo, as¡ que si el argumento opcional -n est dado, ordenar las l¡neas de entrada en forma num,rica en vez de alfab,tica.

Un sort consiste, frecuentemente en tres partes - una "comparaci¢n" que determina el orden de alg£n par de objetos, un "intercambio" que invierte su orden, y un "algoritmo de ordenaci¢n" que compara e intercambia hasta que los objetos estn en orden. El algoritmo de ordenaci¢n es independiente de las operaciones de comparaci¢n e intercambio , as¡ pasando diferentes funciones de comparaci¢n e intercambio, podemos arreglar para sortear por diferentes criterios. Este es el acceso tomado en nuestro nuevo sort.

La comparaci¢n lexicogrfica de dos l¡neas es hecha por strcmp e intercambiada por swap ,como antes; tambi,n necesitaremos una rutina numcmp que compara dos l¡neas (valor num,rico) y devuelve la indicaci¢n
del mismo g,nero de la condici¢n, como lo hace strcmp. Estas tres funciones son declaradas en main y los punteros para ellos son pasados a
sort. sort en cambio llama las funciones v¡a punteros. Hemos escatimado en errores de proceso para argumentos, as¡ como concentrarse
en el problema principal.

```
define LINES   100      /* maximo numero de lineas a
                           ordenar */
main(argc, argv)        /* ordena lineas de entrada */
int argc;
char *argv  ;

    char *lineptr LINES ;    /*punteros para lineas de
                               texto */
    int  nlines;        /* numero de lineas leidas en
                           la entrada */
    int  strcmp(), numcmp(); /* funciones de
                                comparacion */
    int  swap();        /* funcion de intercambio */
    int numeric = 0;        /* 1 si el sort es
                               numerico */

    if (argc > 1 && argv 1  0  == '-' && argv 1  1  ==
                  'n')
        numeric = 1;
    if ((nlines = readlines(lineptr, LINES)) >= 0)
        if (numeric)
            sort(lineptr, nlines, numcmp, swap);
        else
            sort (lineptr, nlines, strcmp, swap);
        writelines(lineptr, nlines);
     else
        printf ("entrada muy grande para ordenar n");
```

strcmp, numcmp y swap son direcciones de funciones; ya que ellas son

funciones conocidas, el operador & no es necesario, en la misma forma esto no es necesitado antes en un nombre de arrglo. El compilador arregla la direcci¢n de la funci¢n a ser pasada.

El segundo paso es modificar sort:

```
sort(v, n, comp, exch)   /* ordena strings v 0 ...v n-1
                          */
char *v  ;                /* en orden creciente */
int n;
int (*comp)(), (*exch)();

    int gap, i, j;

    for (gap = n/2; gap > 0; gap /= 2)
        for (i 0 gap; i < n; i++)
            for (j = i-gap; j >= 0; j -= gap)
                if ((*comp)(v j ), v j+gap ) <= 0
                    break;
                (*exch)(&v j ), &v j+gap );
```

Las declaraciones deber¡an ser estudiadas con cuidado.

```
int(*comp)()
```

dice que comp es un puntero para una funci¢n que devuelve un entero. El primer conjunto de par,ntesis son necesarios; sin ellos

```
int *comp()
```

dir¡a que comp es una funci¢n devolviendo un puntero a un entero, que es
una cosa completamente diferente.

El uso de comp en la l¡nea

```
if ((*comp)(v j , v j+gap ) <= 0)
```

es consistente con la declaraci¢n: comp es un puntero para una funci¢n, *comp es la funci¢n y

```
(*comp)(v j , v j+gap )
```

es la llamada para ella. Los par,ntesis son necesarios, as¡ los componentes estn correctamente asociados.

Ya hemos mostrado strcmp, que compara dos strings. Aqu¡ est numcmp, que compara dos strings en un valor num,rico dirigido:

```
numcmp(s1, s2)      /*compara s1 y s2 numericamente */
char *s1, *s2;

    double atof(), v1, v2;

    v1 = atof(s1);
    v2 = atof(s2);
    if (v1 < v2)
        return(-1);
    else if (v1 > v2)
        return(1);
    else
        return(0);
```

El paso final es sumar la funci¢n swap que intercambia do punteros. Esto es adaptado directamente desde la forma que presentamos tempranamente en este cap¡tulo.

```
swap(px, py)        /*intercambia *px y *py */
char *px  , *py  ;

    char *temp;

    temp = *px;
    *px = *py;
    *py = temp;
```

Hay una variedad de otras opciones que pueden ser sumadas al programa de
sorteo.

_

CAPITULO 6 ESTRUCTURAS

Una estructura es una coleccion de una mas variables, posiblemente de diferentes tipos, agrupadas a la vez bajo un solo nombre para una manipulacion conveniente. (las estructuras son llamadas "registros" en algunos lenguajes, mas notablemente en Pascal).

El ejemplo tradicional de una estructura es el registro de una nomina: un "empleado" es descrito por un set de atributos tales como el nombre, direccion, numero de seguro social, salario, etc. Algunas de estas en cambio podrian ser estructuras: un nombre tiene varios componentes, como es una direccion y aun un salario.

Las estructuras ayudan a organizar complicados datos, particularmente en programas largos, porque en muchas situaciones de ellas permiten un grupo de variables relacionadas para ser tratadas como una unidad en vez de entidades separadas. En este **CAPITULO** trataremos
de ilustrar como son usadas las estructuras. Los programas que usaremos son mas grandes que muchos de los otros en este libro.

6.1 BASICOS

Nos permitiremos revisar la conversion de rutinas de datos del cap.5 . Un dato consiste de varias partes, tal como el dia, mes, y ano, y quizas el dia del ano y el nombre del mes. Estas cinco variables pueden todas ser localizadas dentro de una estructura como esta:

```
struct date {
    int day;
    int month;
    int year;
    int yearday;
    int mon_name[4];
};
```

La palabra struct introduce una declaracion de estructura, que es una lista de declaraciones encerradas en llaves. Un nombre opcional, llamado una estructura de rotulo puede seguir a la palabra struct (tal como aqui: date). El rotulo del nombre es de la clase de la estructura, y pueden ser usados subsecuentemente como una taquigrafia para la declaracion detallada.

Las variables o elementos mencionados en una estructura son llamados "miembros". Un miembro de la estructura o rotulo y una variable comun pueden tener el mismo nombre sin problema, ya que que pueden siempre ser distinguidas por contexto. Naturalmente como un asunto de estilo uno normalmente usaria el mismo nombre por supuesto solo relacionado a objetos.

La llave de la derecha que termina la lista de miembros puede estar seguida por una lista de variables, justo como para algun tipo basico. Esto es,

```
struct { ... } x, y, z;
```

es sintacticamente analogo a

```
int x, y, z;
```

en el sentido que cada afirmacion declara a x, y, z por ser variables de el tipo nombrado y causan espacio para ser asignadas.

Una declaracion de estructura que no es seguida por una lista de variables no determina almacenaje; esto meramente describe un patron o el modelo de una estructura. Asi la declaracion es rotulada, sin embargo, el rotulo puede ser usadao mas tarde en difiniciones de intancias actuales de la estructura. Por ejemplo, dar la declaracion del date mas alto.

```
struct date d;
```

define una variable d que es una estructura del tipo date. Una estructura externa o estatica puede estar inicializada por la siguiente definicion con una lista de inicializadores para componentes:

```
struct date d ={ 4, 7, 1776, 186, "Jul" };
```

Un miembro de una estructura en particular es referido a una expresion por una contruccion de la forma

```
structura-nombre . miembro
```

La estructura del operador miembro "." conecta el nombre de la estructura y el nombre del miembro. Para poner el salto desde el dato en la estructura d, por ejemplo,

```
leap = d.year % 4 == 0 && d.year % 100 != 0
        || d.year % 400 == 0;
```

o para chequear el nombre del mes,

```
if (strcmp(d.mon_name, "Aug") == 0) ...
```

o para convertir el primer caracter del nombre del mes a mayuscula,

```
d.mon_name[0] = lower(d.mon_name[0]);
```

Las estructuras pueden estar anidadas; una nomina podria actualmente verse como

```
struct person  {
    char name[NAMESIZE];
    char address[ADRSIZE];
    long zipcode;
    long ss_number;
    double salary;
    struct date birthdate;
    struct date hiredate;
};
```

La estructura person contiene dos datos. Si declaramos emp como

```
struct person emp;
```

entonces

```
emp.birthdate.month
```

se refiere al mes de nacimiento. La estructura del operador mienbro . asocia la izquierda con la derecha.

6.2 ESTRUCTURAS Y FUNCIONES

Hay un numero de restriciones en estrucuras C. Las reglas esenciales son que las operaciones que solo Ud. puede representar en una estructura son tomadas estas direcciones con &, y un acceso de sus miembros. Esto implica que las estructuras no pueden estar asignadas o copiadas como una unidad, y que ellas no pueden ser pasadas o devueltas desde funciones. (Estas restriciones seran removidas en proximas versiones). Punteros para estructuras no sufren estas limitaciones, no obstante, as~ estructuras y funciones hacen a la vez el trabajo confortablemente. Finalmente, estructuras automaticas, como arreglos automaticos, no pueden ser inicializados; s}lo pueden las estrucuras externas o estaticas.

Investigaremos algunos de estos puntos para reescribir el dato de conversion de funciones de el ultimo **CAPITULO** para usar estructuras. Ya que las reglas prohiben pasar de una estructura a una funcion directamente, no deberiamos tampoco pasar los componentes separadamente, o pasar un puntero para todo el asunto. La primera alternativa usa day_of_year como escribimos en el cap.5:

```
d.yearday = day_of_year(d.year, d.month, d.day);
```

La otra manera es de pasar un puntero. Si tenemos declaredo hiredate como

```
struct date hiredate;
```

y re-escrito day_of_yearPB, podemos entonces decir

```
hiredate.yearday = day_of_year(&hiredate);
```

al pasar un puntero hacia hiredate o hacia day_of_year. La funcion tiene que ser modificada porque estos argumentos estan ahora en un puntero mas bien que en una lista de variables

```
day_of_year(pd)   /* cojunto de dias del ano del mes,
                     dia  */
struct date *pd;

    int i, day, leap;

    day = pd->day;
    leap = pd->year % 4 == 0 && pd->year % 100 != 0
           !! pd->year % 400 == 0;
    for (i = 1; i < pd->month; i++)
        day += day_tab leap  i ;
    return(day)
```

La declaracion

```
struct date *pd;
```

dice que pd es un puntero para una estructura del tipo date. La notacion ejemplificada por

```
pd->year
```

es nueva. Si p es un puntero para la estructura, entonces

```
p->miembro-de-estructura
```

se refiere a el miembro en particular. (El operador -> es un signo menos seguido por >).

Ya que pd es un puntero para la estructura, el miembro year puede tambi`n ser referido tal como

```
(*pd).year
```

pero punteros para estructuras son asi frecuentemente usados que la notacion -> es proveida como una taquigrafia conveniente. Los parentesis son necesarios en (*pd).year porque la precedencia del operador de estructura de miembro . es mas alta que *. Ambos -> y . asocian desde izquierda a derecha, asi

```
p->q->memb
emp.birthdate.month
```

son

```
(p->q)->memb
(emp.birthdate).month
```

Para la perfeccion aqui esta la otra funcion, month_day, re-escrita para usar la estructura

```
month_day(pd)   /*   conjunto de meses y dias desde el
                       dia del ano   */
struct date *pd;

    int in, leap;

    leap = pd->year % 4 == 0 && pd->year % 100 != 0
           !! pd->year % 400 == 0;
    pd->day = pd->yearday;

    for (i = 1; pd->day > day_tab leap  i ; i++)
        pd->day -= day_tab leap  i ;

    pd->month = i;
```

Los operadores de estructura -> y ., junto con () para listas de argumentos y [] para subscritos, son para el tope de la precedencia jerarquica. Por ejemplo, dada la declaracion

```
struct
    int x;
    int *y;
  *p;
```

entonces

```
++p->x
```

incrementa x, y no p, porque el parentesis implica ++(p->x). Los parentesis pueden ser usados para alterar: (p++)->x incrementa p antes de accesarce en x, y (p++)->x incrementa p despues. ( ]....?).

6.3 ARREGLOS DE ESTRUCTURAS

Las estructuras son especialmente convenientes para manejar arreglos de variables relacionadas. Por el momento, considerar un programa para contar las ocurrencias de cada keyword de C. Necesitamos un arreglo de string de caracteres para tener los nombres, y un arreglo de enteros para contarlos. Una posibilidad es la de usar dos arreglos keyword y keycount, como en

```
char *keyword NKEYS ;
int  keycount NKEYS ;
```

Pero el mismo hecho que los arreglos son paralelos indican que una organizacion diferente es posible. Cada keyword es realmente un par:

```
char *keyword;
int  keycount;
```

y hay un arreglo de pares. La declaracion de estructura

```
struct key
    char *keyword;
    int  keycount;

  keytab NKEYS ;
```

define un arreglo keytab de estructuras de este tipo, y asigna el almecen para ellos. Cada elemento del arreglo es una estructura. Esto tambi`n puede ser escrito

```
struct key
    char *keyword;
    int keycount;
;

struct key keytab NKEYS ;
```

Ya que la estructura keytab actualmente contiene un conjunto constante de nombres, este es facil para inicializarlo una vez y para todos cuando este es definido. La inicializacion de la estructura es bastante analoga a la ya conocida una vez - La definicion es seguida por una lista de inicializadores encerrados en llaves:

```
struct key
    char *keyword;
    int  keycount;

  keytab  =
    "break", 0,
    "case", 0,
    "char", 0,
    "continue", 0,
    "default", 0,
    /* ... */
    "unsigned", 0,
    "while", 0,
;
```

Los inicializadores son listados en pareja correspondiendo a los miembros de la estructura. Esto seria mas preciso para inicializadores encerrados por cada "fila" o estructura en llaves, como en

```
 "break", 0  ,
 "case", 0  ,
```

pero las llaves internas no son necesarias cuando los inicializadores son variables simples o string de caracteres, y cuando todas estan presente. Como siempre, el compiador computara el numero de entradas
en el arreglo keytab s~ los inicializadores estan presentes y los esten a la izquierda vacios.

El programa contador de keyword empieza con la definicion de keytab. La rutina main lee las entradas por llamar repetidamente una funcion getword que va a buscar el input una palabra. Cada palabra es mirada en keytab con una version de la funcion de busqueda binaria que

escribimos antes. (Las listas de keywords tienen que ser dadas en orden creciente para este trabajo).

```
 define   MAXWORD   20

main()   /* contador C de Keywords   */

    int n, t;
    char word MAXWORD ;

    while ((t = getword(word, MAXWORD)) != EOF)
        if (t == LETTER)
         if ((n == binary(word, keytab, NKEYS)) >= 0)
            keytab n .keycount++;

    for (n = 0; n < NKEYS; n++)
        if (keytab n .keycount > 0)
          printf("%4d %s n",
            keytab n .keycount, keytab n .keyword);



binary(word, tab, n)  /*  encuentra word en tab 0 ...
                        tab n-1   */
char *word;
struct key tab  ;
int n;

    int low, high, mid, cond;

    low = 0;
    high = n-1
    while (low <= high)
      mid = (low+high) / 2;
      if((cond = strcmp(word;tab mid .keyword)) <0 )
        high = mid-1;

      else if (cond >0)
            low = mid+1;
      else
        return(mid);

      return(-1);
```

Mostraremos la funcion getword en un momento; por ahora es suficiente para decir que esto devuelve LETTER en el momento que encuentra una palabra, y copia la palabra en este primer argumento.

La cantidad NKEYS es el numero de keywords en keytab. Aunque debemos contar esto para manejarlo, es un lote facil, y seguro de hacerlo por la maquina, especialmente si la lista esta sometida a cambio. Una posibilidad podria ser al terminar la lista de inicializadores con un puntero nulo, entonces el loop a lo largo de keytab hasta el fin.

Pero esto es mas que necesario, ya que el tamano del arreglo es completamente determinado una vez compilado. El numero de entradas
justo es

```
size of keytab ... size of struct key
```

C proporciona un compilado unario operador llamado sizeof que puede ser
usado para computar el tamano de algun objeto. La expresion

```
sizeof(objeto)
```

admite una cantidad entera para el tamano del objeto especificado. (La magnitud esta dada en unidades sin especificar los llamados "bytes", que
son de la misma magnitud como un char). El objeto puede ser una variable actual o arreglo o estructura, o el nombre de un tipo basico como un int o double, o el nombre de un tipo derivado como una estructura. En nuestro caso, el numero de keywords es la magnitud del arreglo dividida por la magnitud de un arreglo de elemento. Este calculo es usado en una instruccion define para el conjunto de valores de NKEYS:

```
define NKEYS  (sizeof(keytab) / sizeof(struct key))
```

Ahora para la funci}n getword. Tenemos actualmente escrito un getword mas general que es necesario para este programa, pero no es realmente m#s complicado. getword devuelve la proxima "palabra" desde
la entrada, donde un word es un string de letras y digitos que comienzan con una letra, o un caracter simple. El tipo del objeto es devuelto como el valor de una funcion; esto es LETTER si lo tomado es una palabra, EOF para fin de archivo, o un mismo caracter no alfabetico.

```
getword(w, lim)   /* encuentra la pr}xima palabra desde
                     la entrada */
char *w;
int lim;

    int c, t;

    if(type(c = *w++ = getch()) != LETTER)
        *w = ' 0';
        return(c);



    while (--lim > 0)
        t = type(c = *w++ = getch());
        if (t != LETTER && t != DIGIT)
            ungetch(c);
            break;



    *(w-1) = ' 0';
    return(LETTER);
```

getword usa las rutinas getch y ungetch que escribimos en el cap.4 : cuando la coleccion de una senal (token) alfabetica se detiene, getword fue un caracter demaciado lejano. a llamada para un getch empuja el caracter devuelta a la entrada para la proxima llamada.

getword llama a type para determinar el tipo de cada caracter individual de entrada. Aqui esta una version para el ASCII, solo el

Alfabeto.

```
type(c)   /*  devuelve tipo del caracter ASCII  */
int c;


    if (c >= 'a' && c <= 'z' !! c >= 'A' && c <='Z')
        return(LETTER);
    else  if (c >= '0' && c <= '9')
        return(DIGIT);
    else
        return(c);
```

Las constantes simb}licas LETTER y DIGIT pueden tener algunos valores
que no son conflictivos con los caracteres no-alfabeticos y EOF; las elecciones obvias son

```
define   LETTER   'a'
define   DIGIT    '0'
```

getword pede ser fijado si al llamar la funcion type son reemplazados por referencias para un apropiado type . Las bibliotecas estandares de C proporcionan macros llamadas isalpha y isdigit que operan en esta manera.

6.4 PUNTEROS PARA ESTRUCTURAS

Para ilustrar algunas de las consideraciones involucradas con punteros y arreglos de estructuras, nos permitimos escribir el programa contador de keyword otra vez, ahora usando punteros en vez de un arreglo
subindicado.

La declaracion externa de keytab no necesita cambios, pero main y binary hacen necesaria una modificacion.

```
main()   /*  contador C de keywords; version con
                        puntero  */
    int;
    char word MAXWORD ;
    struct key *binary(), *p;

    while ((t = getword(word, MAXWORD)) != EOF)
        if (t == LETTER)
            if ((p=binary(word, keytab, NKEYS))
                        != NULL)

                p->keycount++;
    for (p = keytab; p < keytab + NKEYS; p++)
        if (p->keycount > 0)
            printf("%4d %s n", p->keycount,
                    p->keyword);



struct key *binary(word,tab,n) /* encuentra palabra */
char *word;    /* en tab 0 ...tab n-1  */
struct key tab  ;
int n;

    int cond;
    struct key *low = &tab 0 ;
    struct key *high = &tab n-1 ;
    struct key *mid;

    while (low <= high)
        mid = low + (high-low) / 2;
        if ((cond = strcmp(word,mid->keyword)) < 0)
            high = mid - 1;
        else  if (cond > 0)
            low = mid + 1;
        else
            return(mid);

    return(NULL);
```

Hay varias cosas dignas de notar aqui. Primero, la declaracion de binary debe indicar que le devuelve un puntero a la estructura del tipo key, en lugar de un entero; esto es declarando en ambas main y binary. Si binary encuentra la palabra, devuelve un puntero a esta; si esto falla, le devuelve NULL.

Segundo, todo el accesamiento de elementos de keytab es hecho por punteros. Esto es una causa significativa de cambio en binary: el c#lculo del elemento central no puede ser de largo simple

```
mid = (low+high) / 2
```

porque la suma de dos punteros no producira alguna bondad de una respuesta util (aun cuando sea dividido por 2), y en realidad es ilegal. Esto debe ser cambiado por

```
mid low + (high-low) / 2
```

en el cual mid apunta al elemento equidistante entre low y high.

Ud. deberia tambien estudiar los inicializadores para low y high. Esto es posible para inicializar un puntero para las direcciones de un objeto con anterioridad definido; que es precisamente lo que tenemos que
hacer aqui.

En main escribimos

```
for (p = keytab; p< keytab + NKEYS; p++)
```

Si p es un puntero para una estructura, cualquier aritmetica con p toma en relacion el lugar actual de la estructura, asi p++ incrementa p por la correcta cantidad para tomar el proximo elemento del arreglo de estructuras. Pero no asume que el largo de una estructura es la suma de las longitudes de estos nombres - porque de requerimientos alineados para diferentes objetos, all~ pueden estar "perforado" en la estructura.

Finalmente, un aparte en el formato del programa. Cuando una funcion devuelve un tipo complicado, como en

```
struct key *binary(word, tab, n)
```

El nombre de la funcion puede ser dificil de ver, y para encontrar con un texto editor. Por consiguiente un estilo alternativo es algunas veces usado :

```
struct key *
binary(word, tab, n)
```

Esto es mayormente un asunto de gusto personal; escoga la forma que Ud.
guste y dominelo.

6.5 ESTRUCTURA REFERENCIAL POR SI MISMA

Queremos suponer para manejar el problema mas general de contar las
ocurrencias de todas las palabras en alguna entrada. Ya que las listas de palabras no son conocidas en avance, no podemos convenientemente sortearlas y usar una busqueda binaria. Todav^a no podemos hacer busqueda lineal para cada palabra como ellas arrivan, para ver si `stas

han sido vistas; el programa tomar^a siempre. (M#s precisamente, su esperado funcionamiento se desarrollar~a cuadricalmente con el n^mero de
palabras en la entrada.)]C}mo podr~amos organizar el dato para copiar
eficientemente con una lista de palabras arbitrarias?.

Una soluci}n es la de guardar el conjunto de palabras v~stas hasta ahora sorteadas en toda momento, para localizar cada palabra hasta su propia posici}n en el orden como arriban. Esto no podr~a ser hacho para trasladar palabras en un arreglo lineal, sin embargo usar~amos un dato de estructura llamado un #rbol binario.

El #rbol contiene un "nodo" por palabra ; cada nodo contiene

un puntero para el texto de la palabra
una cuenta del n^mero de ocurrencias
un puntero para el nodo hijo izquierdo
un puntero para el nodo hijo derecho

El nodo no puede tener m#s que dos hijos; `ste podr~a tener s}lo un 0 o un 1.

Los nodos son mantenidos as~ que cualquier nodo el sub#rbol izquierdo contiene s}lo palabras que son menores que la palabra en el nodo, y el sub#rbol derecho contiene s}lo las palabras m#s grandes. Para descubrir si una nueva palabra ya esta en el #rbol, en un comienzo en el origen y comparar la nueva palabra con la palabra almacenada en el
nodo. S~ ellos concuerdan, la pregunta es respondida afirmativamente. Si la nueva palabra es menor que la palabra del #rbol, la busqueda hasta el hijo izquierdo; de otro modo el hijo derecho es investigado. Si no hay hijo en la direcci}n requerida, la nueva palabra no est# en el #rbol, y en efecto el propio place gor it to be is missing child.
Este busca procesos que estan inherentemente recursivos, ya que la busqueda desde alg^n nodo usa una busqueda de sus hijos. Por consiguiente, rutinas recursivas para inserci}n e impresi}n ser#n m#s natural.

Partiendo hacia la descripci}n de un nodo, este es claramente una estructura con 4 componentes.

```
struct tnode        /*  el nodo b#sico  */
    char *word;    /*  puntero para el texto  */
    int count;     /*  n^mero de ocurrencias  */
    struct tnode *left;      /*  hijo izquierdo  */
    struct tnode *right;     /*  hijo derecho  */
;
```

Esta declaraci}n "recursiva" de un nodo puede mirarse por suerte, pero esto es actualmente bastante correcto. Esto es ilegal para una estructura que contiene una misma instancia, pero

```
struct tnode *left;
```

declara a left para ser un puntero hasta un nodo, no un mismo nodo.

El c}digo para todo el programa es sorpresivamente peque|o, dado un manojo de rutinas sostenidas que ya hemos escrito. Estas son getword,para ir a buscar cada palabra de entrada, y abastecer espacio para squirreling las palabras ausentes.

La rutina main simplemente lee palabras con getword y las instala en el #rbol con tree.

```
 define  MAXWORD   20
main()        /*  cuenta frecuencia de la palabra  */

    struct tnode *root, *tree();
    char word MAXWORD ;
    int t;

    root = NULL;
    while ((t = getword(word, MAXWORD)) != EOF)
        if (t == LETTER)
            root = tree(root, word);
    treeprint(root);
```

El mismo tree es recto. Una palabra es presentada por main para el tope del nivel del #rbol. En cada etapa, la palabra es comparada con la palabra ya almacenada en el nodo, y es filtrada abajo hacia el sub#rbol de la izquierda o la derecha por un llamado recirsivo hacia el #rbol. Eventualmente la palabra marca algo en el #rbol (en que caso la cuenta es incrementada), o un puntero nulo es encontrado, indicando que
un nodo debe ser creado y agregado al #rbol. Si un nuevo nodo es creado, tree devuelve un puntero para `l, que es instalado en el nodo padre.

```
struct tnode *tree(p, w)   /*  intala w en p o abajo de
                               p  */
struct tnode *p;
char *w;


    struct tnode *talloc();
    char *strsave();
    int cond;

    if (p == NULL)      /*   una nueva palabra ha
                             arrivado  */
        p = talloc();   /*  fabrica el nuevo nodo  */
        p->word = strsave(w);
        p->count = 1;
        p->left = p->right = NULL;
      else if ((cond = strcmp(w, p->word)) == 0)
            p->cound++;    /* palabra repetida  */
    else if (cond < 0)   /*  el menor se dirige al
                             sub#rbol izquierdo  */
            p->left = tree(p->left, w);
    else    /* el m#s grande al sub#rbol derecho  */
        p->right = tree(p->right, w);
    return(p);
```

El almacen para el nuevo nodo es buscado por la rutina talloc, que es una adaptaci}n de la alloc que escribimos antaeriormente. Esto devuelve un puntero, para un espacio libre apropiado para que posea el nodo del #rbol. (Discutiremos esto m#s en un momento.) La nueva palabra es copiada hacia un espacio oculto por strsave, la cuenta es inicializada, y los dos hijos son hechos nulos. Esta partedel c}digo es ejecutada s}lo en el borde del #rbol, cuando un nuevo nodo esta siendo agregado. Hemos (imprudencia para la producci}n del programa)
omitido el error chequiado sobre valores devueltos por strsave y talloc.

treeprint imprime el #rbol en el orden del sub#rbol izquierdo; en cada nodo, esto imprime el sub#rbol izquierdo (todas las palabras menores que esta palabra), entonces la misma palabra, luego el sub#rbol dercho (todas las palabras m#s grandes). Si Ud. siente d`bil acerca de la recursi}n, usted mismo dibuje un #rbol e imprimalo con treeprint; esta es una de las rutinas de recursividad limpia que Ud. puede encontrar.

```
treeprint(p)   /* imprime el #rbol p recursivamente */
struct tnode *p;

    if (p != NULL)
        treeprint(p->left);
        printf("%4d %s n", p->count, p->word);
        treeprint(p->right);
```

Una practica nota: si el #rbol llega a ser "inbalanceado" porque las palabras no han arrivado en orden random, el instante en que corre el programa puede tambi`n desarrollarse r#pido. Como un p`simo caso, si las palabras estan ya en orden, este programa hace una simulaci}n expanciva de b^squeda lineal. Hay generaciones del #rbol binario, notablemente 2-3 #rboles y #rboles AVL, que no hacen soportar esta conducta desde estos p`simos casos, pero no lo descubriremos aqu~.

Antes nos permitiremos este ejemplo, es tambi`n v#lido una resumida
disgresi}n sobre un problema relacionado con el indicador de almacenaje.
Claramente es deseable que all~ est` s}lo un indicador de almacenaje en
un programa, aunque estas diferentes clases de se|ales de objetos.
Pero si un se|aloador es para procesos requeridos por, decir, punteros para char's y punteros para struct tnode's, dos preguntas surgen.
Primero ahora hacerle encontrar el requerimiento de las m#s reales m#quinas esos objetos de ciertos tipos deben satisfacer las restricciones alineadas](por ejemplo, enteros a menudo deben ser se|alados sobre incluso direcciones)?. Segundo,]qu' declaraciones pueden copiarcon el hecho que alloc necesariamente devuelve diferentes clases de punteros?.

Requerimientos alineados pueden generealmente ser satisfechos f#cilmente, a el costo de alg^n espacio desperdiciado, meramente por asegurar que el se|alaodr siempre devuelva un puntero que encuentra todas las restricciones alineadas. Por ejemplo, sobre el PDP-11 es suficiente que alloc siempre devuelva hasta un puntero, ya que cualquier tipo de objeto puede ser almacenado en una direcci}n constante. El s}lo costo es un caracter desperdiciado en una rara longitud solicitada.
Acciones similares son tomadas en otras m#quinas. De otro modo la implementaci}n de alloc no puede ser portable, pero la utilidad est#.
La instrucci}n alloc del cap.5 no hace garant^a de alg^n alineamiento en particular; en el cap.8 mostraremos que hace el trabajo derecho.

La pregunta del tipo de declaraci}n para alloc es una dificultad para cualquier lenguaje que toma `ste tipo chequeando seriamente. En C, el procedimiento }ptimo es para declarar que alloc devuelve un puntero para char, luego expl^citamente obliga el puntero hacia el deseadotipo con un molde. Esto es, si p es declarado como

```
char *p;
```

entonces

```
(struct tnode *) p
```

lo convierte en un puntero tnode en una expresi}n. As~ talloc es escrito como

```
struct tnode *talloc()

  char *alloc();

  return((struct tnode *) alloc(sizeof(structtnode)));
```

Esto es m#s que necessario para compiladorse corrientes, pero representa
la seguridad en curso para el futuro.

6.6 TABLA Lookup

En esta secci}n escribiremos the innards de la tabla lookup paquete como una ilustraci}n de mas aspectos de estructuras. Esta clave es t~pica de que puede ser encontrado en el s~mbolo de la tabla manejada en rutinas de un procesador macro o un compilador. Por ejemplo, considerar la instrucci}n C define.
Cuando una linea como

```
define  YES  1
```

es encontrada, el nombre YES y el reemplazo de texto 1 son almacenados
en una tabla. M#s tarde, cuando el nombre YES aparece en une instrucci}n como

```
inword = YES;
```

esto debe ser reemplazado por 1.

Hay dos grandes rutinas que manipulan los nombres y reemplazn el texto. install(s, t) guarda el nombre s y reemplaza el texto t en una tabla; s y t son justo string de caracteres. lookup(s) b^sca a s en la tabla, y devuelve un puntero en el lugar donde fu` encontrado, o NULL si
`ste no est# all~.

El algor~tmo usado es una minuciosa b^squeda - el entrante nobre es convertido en un peque|o entero positivo, que es luego usado para hacer un ~ndice en el arreglo de punteros. Un arreglo de elemtos apunta al comienzo de una serie de bloques describiendo nombres que tienen ese valor desmesurado. Esto es NULL si no tienen nombres desmesurados para ese valor.

Un bloque en la serie es unsa estructura que contiene punteros para el nombre, al reemplazar el texto, y el nuevo bloque en la serie. Un pr}ximo puntero nulo marca el fin de la serie.

```
struct nlist     /* tabla b#sica de entrada */
    char *name;
    char *def;
    struct nlist *next;  /* pr}xima entrada en la
                            serie */
;
```

El arreglo puntero es justo

```
 define   HASHSIZE   100
static struct nlist *hashtab HASHSIZE ; /* puntero de
                                la tabla */
```

La funci}n hashing, que es usada por ambos lookup y instanll, simplemente suma los valores de los caracteres en el sting y forma las sobras en el m}dulo del largo del arreglo. (Esto no es el mejor algor~tmo posible, pero tiene el merito de una simplicidad extrema).

```
hash(s)   /* forma el valor hash para el string s */
char *s;

    int hashval;

    for (hashval = 0; *s != ' 0'; )
        hashval += *s++;
    return(hashval % HASHSIZE);
```

El proceso hashing produce un ^ndice en marcha en el arreglo hashtab; si el string es para ser encontrado en alg^n lugar, `sto ser# en la serie de los bloques que comienzan all~. La b^squeda es desempe|ada por lookup. Si lookup encuentra la entrada ya presente, devuelve un puntero para `l; si no, lo devuelve NULL.

```
struct nlist *lookup(s)  /* mira para s en hashtab */
char *s;
```

```
struct nlist *np;

for(np = hashtab hash(s) ; np != NULL;
            np = np->next)

    if(strcmp(s, np->name)  == 0)
        return(np);   /* lo encontr} */
return(NULL);   /* no encontrado */
```

install usa a lookup para determinar si el nombre que est# siendo instalado ya est# presente; si es as~, la nueva definici}n debe reemplazar a la antigua. De otro modo, una nueva entrada completamente
es creada. install devuelve a NULL si por alguna raz}n no hay espacio para la nueva entrada.

```
struct nlist *install(name, def)
char *name, *def;     /* en hashtab */

    struct nlist *np, *lookup();
    char *strsave(), *alloc();
    int hashval;

    int((np = lookup(name)) == NULL) /* no encontrado */
        np = (struct nlist *) alloc(sizeof(*np));
        if (np == NULL)
            return(NULL);
        if((np->name = strsave(name)) == NULL)
            return(NULL);
        hashval = hash(np->name);
        np->next = hashtab hashval ;
        hahstab hashval  = np;
      else      /* ya aqu~ */
        free(np->def); /* definici}n previa libre */
    if ((np->def = strsave(def)) == NULL)
        return(NULL);
    return(np);
```

strsave meramente copia el string dado por estos argumentos en un lugar seguro, obtenido por una llamada a alloc. Mostramos el c}digo en
el cap.5. Cada llamada a alloc y free puede ocurrir en alg^n orden, y cada asunto alineado, la versi}n simple de alloc en el cap.5 no es adecuada aqu~; vea los cap.7 y 8.

6.7 CAMPOS

Cuando un espacio almacenado est# en prima, puede ser necesario para empaquetar varios objetos dentro de una simple palabra de m#quina;
un especial y com^n uso es un set de single-bit flags en aplicaciones como compilar tablas de s~mbolos. Externamente impuesto el formato de
datos, tal como interfaces para hardware artificiales, tambi`n frecuentemente requiere la habilidad para tomar en piesas una palabra.

Imagine un fragmento de un compilador que manipula una tabla de s~mbolos: Cada identificador en un programa tiene cierta informaci}nasociada con esto, por ejemplo, si o n `sto es una Keyword, si o no `sto es external y/o static. La manera m3s completa para codificar tal informaci}n es un set de one-bit flags en un simple char o int.

La manera usual esta es hecha para definir un set de "marcas" correspondiendo a la relevante posici}n del bit, como en

```
define   KEYWORD   01
define   EXTERNAL  02
define   STATIC    04
```

(Los n^meros deben ser de dos sifras). Entonces accesando los bits llega a ser un asunto de "bit-fiddling" con el cambio, encubrimiento, y completando operadores que descubrimos en el cap.2.

Ciertos modismos aparecen frecuentemente :

```
flags != EXTERNAL ! STATIC;
```

enciende la EXTERNAL y STATIC bits en flags, mientras que

```
flags &=  (EXTERNAL ! STATIC);
```

vuelve a apagarlo, y

```
if ((flags & (EXTERNAL ! STATIC)) == 0)  ...
```

es verdadero si ambos bits estan off.

C ofrece la capacidad de definir y accesar campos adentro de una palabra directamente m#s bien que por operadores l}gicos bitwise. Un campo es un set de bits adyacentes dentro de un simple int. La sint#xis de una definici}n de campo y acceso esta basado sobre estructuras. Por ejemplo; el simbolo de tabla define's anterior ser~a reemplazada por la definici}n de #rboles de campos :

```
struct
    unsigned  is_keyword : 1;
    unsigned  is_extern : 1;
    unsigned  is_static : 1;
  flags;
```

Esto define una variable llamada flags que contiene un #rbol de campo 1-bit. El n^mero que sigue a los dos puntos representa el campo con anchura en bits. Los campos son declarados unsigned para enfatizar que
ellos realmente son cantidades no se|aladas.

Individualmente los campos son refernciados como flags.is_word, flags.is_extern, etc, justo como otros mienbros de la estructura. Los campos se portan como peque|os, enteros no se|alados, y pueden participar en expresiones aritm`eticas como otros enteros. As~ los ejemplos previos pueden ser escritos m#s naturalmente como

```
flags.is_extern = flags.is_static = 1;
```

para abrir los bits;

```
flags.is_extern = flags.is_static = 0;
```

para cerrarlos; y

```
if ((flags.is_extern == 0 && flags.is_static == 0) ...
```

para consultarlos.

Un campo no puede cruzar sobre un int l~mite; si la anchura causar~a esto para sucedr, el campo es alineado al pr}ximo int l~mite. Los campos no necesitan ser nombrados; los campos no nombrados son usados para rellenar. El ancho especial 0 puede ser usado para forzar alineamiento al pr}ximo int l~mite.

Hay un n^mero de advertencias que aplicar a los campos. Tal vez lo m#s significante, es que los campos son asignados de izquierda adercha en algunas m#quinas y en otras de derecha a izquierda, reflejando la naturaleza de diferentes hardware. Esto significa que aunque los campos son bastantes ~tiles para mantener internemte una estructura de datos, la cuesti{n de que el fin llega primero tiene que

ser cuidadosamente considerado cuando escoga aparte extamente un dato
definido.

Otras restricciones para tener presente: los campos son no se|alados; ellos pueden ser almacenados s}lo en int's (o, equivalentemente, unsigned's); ellos no son arreglos; no tienen direcciones, as~ el operador & no puede ser aplicado a ellos.

6.8 UNIONES

Una union es una variable que puede ocupar (en diferentes momentos)
objetos de diferentes tipos y tama¤os, con el compilador preserva la pista del tama|o y alineamientos requeridos. Las unions proporcionan una manera de manipular diferentes clases de datos en una simple area de
almacen, sin encajar en alguna m#queina dependiente una informaci}n en
el programa.

Como un ejemplo, ora vez desde un compilador tablas de s~mbolos, supone que constantes pueden ser int's, float's o punteros de caracteres. El valor de una constante en particular debe ser almacenada en una variable del mismo tipo, sin embargo esto es m#s conveniente para manejar una tabla si el valor ocupa la misma cantidad de almacenaje y es almacenado en el mismo lugar sin tomar en consideraci}n de este tipo. Este es el prop}sito de una union - para proveer una simple variable que puede letimamnete ocupar alguna vez varios tipos. Como con los campos, la sint#xis est# basada en estructuras.

```
union u_tag
    int ival;
    float Fval;
    char *pval;
 uval;
```

La variable uvalser# bastante grande para ocupar el largo del tipo del arbol, sin tomar en consideracion dejando de lado al m#quina en que esta siendo compilada - el codigo es independiente de las caracteristicas del hardware. Algunos de esos tipos pueden ser asignados a uval y usarlos en expresiones, de este modo los largos como el uso es consistente: el tipo recuperado debe ser el tipo mas recientemente almacenado. Esta es la responsabilidad del programador para guardar el rastro de que tipo es corrientemente almacenado en una union; los resultados son dependientes de la maquina si alguna cosa es almacenada con un tipo y extraido como otra.

Sintacticamente los miembros de una union son accesados como

```
union-nombre.miembro
```

o

```
union-puntero->miembro
```

justo como para estructuras. Si la variable utype es usada para guardar el tipo en curso almacenado en uval, entonces uno podria ver el codigo tal como

```
if (utype == INT)
        printf ("%d\n", uval.ival);
else if (utype == FLOAT)
    printf ("%f n", uval.fval);
else if (utype == STRING)
    printf ("%s n", uval.pval);
else
    printf ("mal tipo %d en utype n ", utype);
```

Las uniones pueden ocurrir dentro de estructuras y arreglos y viceversa. La notacion para accesar un miembro de una union en una estructura (o viceversa) es id`ntica a la estructura aninada. Por ejemplo, en la estructura arreglo definida por

```
struct
    char *name;
    int  flags;
    int  utype;
    union
        int   ival;
        float fval;
        char *pval;
     uval;
  symtab NSYM ;
```

la variable ival es referida tal como

```
symtab i .uval.ival
```

y el primer caracter del string pval por

```
*symtab i .uval.pval
```

En efecto, una union es una estructura en que todos los miembros tienen un equivalente en cero, la estructura es bastante grande para ocupar la "anchura" miembro, y el alineamiento es apropiado para todos los tipos en la union. Como con estructuras, las operaciones corrientes permitidas en uniones solo son accesibles a miembros y toman
las direcciones; las uniones no pueden estar asignadas a, pasadas a funciones, o devueltas por funciones. Los punteros para uniones pueden
ser usados en una manera identica para punteros de estructuras.

El almacanaje se¤alado en el cap.8 muestra como una union puede ser
usada para forzar una variable a ser alineada en una particular clase de almacenaje limite.

6.9 Typedef

C proporciona una facilidad llamada typedef para crear los nombres de un nuevo tipo de dato. Por ejemplo, la declaracion

```
typedef int LENGTH;
```

hace el nombre LENGTH como un sinonimo para int. El "tipo" LENGTH
puede ser usado en declaraciones, casts, etc., en exactamente las mismas maneras que el tipo int puede ser:

```
LENGTH    len, maxlen;
LENGTH    *lengths  ;
```

Similarmente, la declaracion

```
typedef char *STRING;
```

hace al STRING un sinonimo para char * o un caracter puntero, que puede
entonces ser usado en declaraciones como

```
STRING p, lineptr LINES , alloc();
```

Notece que el tipo a sido declarado en un typedef aparece en la posicion de un nombre de variable, no a la derecha despues de la palabra typedef. Sintacticamente, typedef es como la clase de almacen extern, static, etc. Habremos usado tambien mas adelante casos con letras para enfatizar los nombres.

Como un ejemplo mas complicado, podriamos hacer typedef's para el arbol de nodos mostrado en este **CAPITULO** :

```
  typedef struct tnode          /* el nodo b#sico */
 char *word;    /* apunta al texto */
 int  count;    /* n^mero de ocurrencias */
 struct tnode *left;      /* hijo izquierdo */
 struct tnode *right;     /* hijo derecho */
TREENODE, *TREEPTR;
```

Esto crea dos nuevos tipos de keywords llamados TREENODE (una estructura) y TREEPTR (un puntero para la estructura). Entonces la rutina talloc podr~a llegar a ser

```
TREEPTR talloc()

    char *alloc();

    return((TREEPTR) alloc(sizeof(TREENODE)));
```

Debe ser enfatizado que una declaracion typedef no crea un nuevo tipo en algun sentido; meramente suma un nuevo nombre para algun tipo
existente. Tampoco estan alli alguna nueva sematica
: variables declaradas de esta manera tienen exactamente las mismas propiedades como las variables cuyas declaraciones son deletreadas explicitamente. En efecto, typedef es como define, excepto que despues de esto es interpretado por el compilador, pueden copiar con sustituciones textuales que estan por encima de las capacidades del preprocesador macro de C. Por ejemplo,

```
typedef int (*PFI)();
```

crea el tipo PFI, puntero "para la funcion que devuelve enteros", que puede ser usado en contexto como

```
PFI strcmp, numcmp, swap;c
```

en el sorteo del cap.5.

Hay dos principales razones para usar declaraciones typedef. La primera es para parametrizar un programa en contraste con la portabilidad de programas. Si typedef's son usadas para tipos de datos que pueden ser dependientes de la maquina, el typedef
solo necesita cambiarse cuando el programa es movido. Una situacion comun es que usar nombres typedef para diferentes cantidades enteras, luego hacer un apropiado set de alternativas para short, int y long para cada m#quina.

El sendo proposito de typedef es para proveer mejor documentacion para un programa - un tipo llamado TREEPTR puede ser facil para entender
que una declarada solo como un puntero para una estructura compilada.

Finalmente, hay siempre la posibilidad que en el futuro el compilador o algun otro programa tal como lint puede hacer uso de la informacion contenida en declaraciones typedef para ejecutar alguna revisada extra de un programa.

_

CAPITULO 7 INPUT Y OUTPUT

Las facilidades de I/O no son parte del lenguaje C, por lo tanto habemos enfatizado entonces en nuestra presentacion hasta aqui. No obstante, programas reales hacen interactuar con su medio ambiente en muchas mas complicadas maneras que aquellas que hemos mostrado antes.

En este **CAPITULO** describiremos "la biblioteca standard de I/O", un conjunto de funciones asignadas para proveer un sistema standard de I/O
para programas C. Las funciones son destinadas a presentar una conveniente interface, sin embargo solo refleja operaciones que pueden ser abastecidas sobre la mayor parte de modernos sistemas operativos. Las rutinas son bastantes eficientes que usarlas raramente sentirian la necesidad de enga|arlo "por eficiencia" sin tomar en consideracion como criticar la aplicacion. Finalmente, las rutinas son significantes para ser "portables", en el sentido que ellas existiran en forma compatible en algun sistema donde exista C, y esos programas que limitan su sistema
de interacciones para facilitar lo abastecido por la biblioteca standard puede ser movido desde un sistema a otro esencialmente sin cambio.

No trataremos de describir la biblioteca integra de I/O aqui; estamos mas interesados en mostrar las escrituras esenciales de programas C que interactuan con su medio y el sistema operativo.

7.1 ACCESO A LA BIBLIOTECA STANDARD

Cada archivo fuente que se refiere a una funcion de biblioteca standard que debe contener la linea.

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

proxima al comienzo. El archivo stdio.h define ciertas macros y variables usadas por la biblioteca de I/O. El uso de los parentesis de angulos < y > en lugar de las comillas dobles usuales dirigen el compilador para buscar por el archivo en un directorio conteniendo una informacion standard a la cabeza (en UNIX, tipicamente lusrlinclude).

Ademas puede ser necesario luego que al cargar el programa a especificar en la biblioteca explicitamente; por ejemplo, en el sistema UNIX de PDP-11, el comando para compilar un programa seria

```
cc archivos fuentes, etc.  -ls
```

donde -ls indica cargar desde la biblioteca standard. (el caracter l es la tetra ele).

7.2 I/O STANDARD - Getchar y Putchar

El mecanisme de entrada simple es para leer un caracter en el instante desde la "entrada standard", generalmente lo usa el terminal, con getchar. getchar() devuelve el proximo caracter de entrada en cada

instante que es llamado. En medios que m#s apoya C, un archivo puede
ser sustituido por el terminar junto con usar la conversion < : s~ un programa prog usa getchar, entonces la linea de comando

```
prog <infile
```

causa que prog lea infile em lugar del terminal. Los switching de la entrada estan hechos de tal manera que el prog mismo es olvidado por el cambio; en particular, el string "<infile" no esta incluido en el argumento de la linea de comando en argv. La entrada de switching es tambien invisible si la entrada viene desde otro programa via una llamada de mecanismo; la linea de comando

```
otherprog | prog
```

corre los dos progranas otherprog y prog, y arregla que la entrada standard para prog venga desde la salida standard de otherprog.

getchar devuelve el valor de EOF cuando lo encuentra en el fin de archivo en cualquier entrada que esta siendo leida. La biblioteca standard define la constante simbolica EOF para ser -1 (con un #define en el archivo stdio.h), pero la pregunta seria escrita en terminos de EOF, y no -1, asi es como es independiente del valor especifico.

Para la salida, putchar(c) pone el caracter c en la "salida standard", que es tambien por omision el terminal. La salida puede ser dirigida a un archivo usando al lado > : si prog usa putchar,

```
prog >outfile
```

escribira la salida standard sobre outfile en lugar del terminal. En el sistema UNIX, en una llamada puede tambien ser usado:

```
prog | anotherprog
```

pone la salida standard de prog dentro de la entrada standard de otherprog. Otra vez, prog no es enterado de la redireccion.

La salida producida por printf tambien encuentra la manera para la salida standard, y llama a putchar y printf que pueden ser interfoliados.

Un sorprendente numero de programas leen solo una entrada en curso
y escriben solo una salida en curso; para tales programas de I/O con getchar, putchar, y printf pueden ser enteramente adecuada, y es ciertamente bastante para tomar lo empezado. Esto es particularmente verdadero dado la redireccion del archivo y una llamada facil para conectar la salida de un programa a la entrada del proximo. Por ejemplo, considerar el programa lower, que indica su entrada para el caso mas inferior :

```
#include  <stdio.h>

main()  /* convierte la entrada al caso mas inferior */
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(isupper(c) ? tolower(c) : c);
}
```

Las funciones issuper y tolower son actualmente macros definidas en stdio.h. La macro issuper pregunta si su argumento es una letra mayuscula, devolviendo no-cero si es, y cero si no. La macro tolower convierte una letra mayuscula a minuscula. Sin tomar en cuenta como esas funciones son implementadas en una maquina en particular, su comportamiento exterior es el mismo, asi los programas que lo usan estan
protegidos desde el conocimiento del set de caracter.

Para convertir multiples archivos, Ud. puede usar un programa como el utilitario cat de UNIX, para llamar los archivos:

```
cat file1 file2...!lower >output.
```

y asi evitar aprender como accesar archivos desde un programa. (cat es presentado al final del **CAPITULO**.)

Como un apartado, en la biblioteca de I/O las funciones getchar y putchar pueden actualmente ser macros, y asi evitar el encabezamiento de
una funcion llamada por caracter. Mostraremos como se hace esto en el cap.8.

7.3 FORMATEO DE LA SALIDA - PRINTF

Las rutinas printf para el output y scanf para el input (proxima seccion) permite una traslacion hacia y desde representaciones de cantidades numericas. Ellas tambien permiten una generacion o interpretacion de lineas formateadas. Hemos usado printf informalmente
en todos los **CAPITULO**s previos; esta es una descripcion mas completa y
precisa.

```
printf(control, arg1, arg2, ...)
```

printf convierte, formatea e imprime sus argumentos en la salida standard bajo el control del string control. El string de control contiene dos tipos de objetos: caracteres comunes, que son simplemente copiados a la salida corriente, y especificaciones de conversion, cada uno de los cuales provoca la conversion e impresion del proximo argumento sucesivo para printf.

Cada especificacion de conversion es introducida por el caracter % y finalizado por una conversion del caracter. Entre el % y la conversion de caracter puede haber:

- Un signo menos, que especifica un ajuste en la izquierda del argumento convertido en este campo.
- Un digito de string especificando un campo de anchura minimo. El numero convertido sera impreso en un campo por lo menos de su ancho, y se ensanchara si es necesario. Si el argumento convertido tiene menos caracteres que el ancho del campo este sera rellenado a la izquierda (o derecha si el indicador de ajuste izquierdo ha sido dado) para incrementar el ancho del campo. El caracter relleno es normalmente un blanco y cero si el ancho del campo fue especificado con un cero delante (este cero no implica un ancho de campo octal.)
- Un periodo, que separa el ancho del campo desde el proximo digito string .
- Un digito string (la precision), que especifica el numero maximo de carcteres para ser impresos desde un string, o el numero de digitos a ser impresos a la derecha del punto decimal de un float o double.
- Un modificador de longitud l (letra ele), que indica que el correspondiente item de datos es un long masbien que un int.

La connversion de caracteres y su significado son:

d El argumento es convertido a notacion decimal.
o El argumento es convertido a notacion octal no se|alada (sin un cero delante).
x El argumento es convertido a notacion hexadecimal no se|alada (sin un ox delante).
u El argumento es convertido a notacion decimal no se|alada.
c El argumento es tomado para ser un simple caracter.
s El argumento es un string; los caracteres del string

son impresos hasta que un caracter nulo es alcanzado o hasta que el numero de caracteres indicados por la especificacion de precision exhaustivo.

e El argumento es tomado para ser un float o double y convertido a notacion decimal de la forma - m.nnnnnnE + xx donde la longitud del string de n's es especificado por la precision. La falta de precision es 6.

f El argumento es tomado para ser un float o double y convertido a notacion decimal de la forma - mmm.nnnnn donde la longitud del string de n'es especificado por la precision. La precision es 6. Notar que la precision no determina el numero de digitos significativos impresos en formato f.

g Usa %e o %f, los ceros no significativos no son impresos.

Si el caracter despues del % no es un caracter de conversion, ese caracter es impreso; asi % puede ser impreso por %%.

Mas conversiones del formato son obvias, y han sido ilustradas en **CAPITULO**s anteriores. La siguiente tabla muestra el efecto de una variedad de especificaciones en la impresion de "hello, world" (12 caracteres). Hemos puesto dos puntos alrededor de cada campo asi Ud. puede verlo en extenso.

```
:%10s:        :hello, world:
:%-10s:       :hello, world:
:%20s:        :       hello, world:
:%-20s:       :hello, world       :
:%20.10s:     :          hello, wor:
:%-20.10s:    :hello, wor         :
:%.10s:       :hello, wor:
```

Una advertencia: printf usa su primer argumento para decidir cuantos argumentos siguen y de que tipo son. Esto traera confusion, y Ud. obtendra respuestas sin sentido, si no hay bastantes argumentos o ellos son de tipo equivocado.

7.4 FORMATEO DE LA ENTRADA - SCANF

La funcion scsnf es la entrada analoga a printf, proporcionando muchas de las mismas facilidades de conversion en la direccion opuesta.

```
scanf(control, arg1, arg2,...)
```

scanf lee caracteres desde la entrada standard, interpretandolas de acuerdo al formato especificado en control, y almacena los resultados en el resto de los argumentos. El argumento de control esta descrito antes; los otros argumentos, CADA UNO DE LOS CUALES DEBE SER UN PUNTERO,
indican donde la correspondiente entrada convertida seria almacenada.

El string de control usualmente contiene especificaciones de control, que son usadas en una interpretacion directa de secuencias de entrada. El string de control puede contener:

- Blancos, tabs o newlines (" espacios en blanco"), que son ignorados.
- Caracteres comunes (no %) que son esperados para marcar el proximo caracter no-blanco de la entrada en curso.
- Especificaciones de conversiones, consistentes del caracter %, un asignamiento opcional es la supresion del caracter *, un numero opcional especificando el ancho maximo (campo), y una conversion de caracter.

Una especificacion de conversion dirige la conversion del proximo campo
de entrada. Normalmente el resultado es ubicado en la variable apuntada por el correspondiente argumento. Si la supresion del asignamiento es indicado por el caracter *, sin embargo, el campo de entrada es simplemente saltado; el asignamiento no es hecho. Un campo
de entrada es definido como un string de caracteres no blancos; extendidos hasta el proximo caracter en blanco o hasta el ancho del campo, si esta especificado, es exhaustivo. Esto implica que scanf leera a traves de la linea limite para encontrar su entrada, ya que los newlines son espacios en blanco.

La conversion de caracter indica la interpretacion del campo de entrada; el argumento correspondiente debe ser un puntero, segun lo requeridopor la llamda del valor semantico de C. Las siguientes conversiones de caracteres son legales:

d Un entero decimal es esperado en la entrada; el argumento correspondiente seria un puntero entero.

o Un entero (con o sin un cero delante) es esperado en la entrada; el argumento correspondiente seria un puntero entero.

x Un entero hexadecimal (con o sin un 0x delante) es esperado en la entrada; el argumento correspondiente seria un puntero entero.

c Un caracter simple es esperado; el argumento correspondiente seria un caracter puntero; el proximo caracter de entrada es situado en el lugar indicado. El salto normal sobre caracteres en blanco es suprimido en este caso; para leer el proximo caracter no blanco, use %1s.

s Un caracter de string es esperado; el argumento correspondiente deberia ser un caracter puntero apuntando a un arreglo de caracteres bastante grande para aceptar el string y una terminacion \0 que sera sumado.

f Un numero flotante es esperado; el argumento correspondiente debe ser un puntero para un float. El caracter de conversion e es un sinonimo para f. El formato de entrada para float's es un signo opcional, un string de numeros posiblemente conteniendo un punto decimal, y un campo
opcional para el exponente conteniendo una E o una e seguida por un posible entero senalado.

Los caracteres de conversion d, o y x pueden ser precedidos por l (letra ele) para indicar que un puntero para long mas bien que un int aparece en la lista de argumento. Similarmente, los caracteres de conversion e o f pueden ser precedidos por l para indicar que un puntero para un double esta en la lista de argumento en vez de un float.

Por ejemplo, el llamado

```
int i;
float x;
char name [50];
scanf("%d %f %s", &i, &x, name);
```

con la linea de entrada

```
25   54.32E-1  Thompson
```

asignara el valor 25 a i, a x el valor 5.432, y el string "Thompson", propiamente terminado por \0, a name. Los tres campos de entrada pueden ser separados por tantos blancos, tabs o newlines como lo desee. El llamado

```
int i;
float x;
char name [50];
scanf ("%2d %f %*d %2s", &i, &x, name);
```

con la entrada

```
56789  0123  45a72
```

asignara 56 a i, 789.0 a x, salta sobre 0123, y ubica el string "45" en name. La proxima llamada para ajguna rutina de entrada comenzara buscando en la letra a. En estos dos ejempos, name es un puntero y asi puede no ser precedido por un &.

Como otro ejemplo, a calculadora rudimentaria del cap.4 ahora puede ser escrita con scanf para hacer la conversion de entrada:

```
#include <stdio.h>

main()    /* calculadora rudimentaria de escritorio */
{
    double sum,v;
    sum = 0;
    while (scanf("%1f", &v) != EOF)
        printf("\t %2f \n", sum += v);
}
```

scanf se detiene cuando agota su string de control, o cuando alguna entrada falla para marcar la especificacion de control. Lo devuelve como su valor el numero de sucesos marcados y asignado a los itemes de
entrada. Esto puede ser usado para decidir cuantos itemes de entradas fueron encontrados. Sobre fin de archivo, es devuelto EOF; note que esto es diferente de cero, lo que significa que el proximo caracter de entrada no mara la primera especificacion en el string de control. La proxima llamada para scanf resume la busqueda inmediatamente despues del
ultimo caracter devuelto.

Una advertencia final: los argumentos para scanf DEBEN ser punteros. El error mas comun es escribir

```
scanf("%d", n);
```

en vez de

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

7.5 FORMATO DE CONVERSION EN MEMORIA

Las funciones scanf y printf tienen siblings llamadas sscanf y sprintf que ejecutan las conversiones correspondientes, pero operan en un string en vez de un archivo. El formato general es

```
sprintf(string, control, arg1, arg2, ...)
sscanf(string, control, arg1, arg2, ...)
```

sprintf formatea los argumentos en arg1, arg2, etc., de acuerdo a control antes visto, pero ubica el resultado en string en vez de la salida standard. El string en curso tiene que ser bastante grande para recibir el resultado. Como un ejemplo, si name es un arreglo de caracter y n es un entero, entonces

```
sprintf(name, "temp%d", n);
```

crea un string de la forma tempnnn en name, donde nnn es el valor de n.

sscanf hace las conversiones inversas - esto escudrina el string de acuerdo al formato en control, y ubica los valores resultantes en arg1, arg2, etc. Estos argumentos deben ser puteros. La llamada

```
sscaf(name, "tenp%d", &n);
```

coloca en n el valor del string de digitos siguientes, temp en name.

7.6 ACCESO DE ARCHIVOS

Los programas escritos hasta ahora, todos han leido la entrada standard y escrito la salida standard, hemos asumido que son magicamente
predefinidos desde un programa, por el sistema operativo local.

El proximo paso en I/O es escribir un programa que accese un archivo que ya no esta conectado al programa. Un programa que ilustra claramente la necesidad para tales operaciones es cat, que concatena un conjunto de archivos nombrados en la salida standard. cat es usado para imprimir archivos en el terminal, y omo un proposito general es un colector de entrada para programas que no tienen la capacidad de accesar
archivos por el nombre. Por ejemplo, el comando

```
cat x.c y.c
```

imprime el contenido de los archivos x.c e y.c en la salida standard.

La pregunta es como ordenar los archivos nombrados, para ser leidos - esto es, como conectar los nombres externos que un usuario piensa de las instrucciones que actualmente leen los datos.

Las reglas son simples. Antes de leer o escribir un archivo este tiene que ser abierto por la funcion de biblioteca, llamada fopen. fopen toma un nombre externo (semejante a x.c o y.c), hace algun quehacer domestico y negociacion con el sistema operativo (detalles que no nos conciernen), y devuelve un nombre interno que debe ser usado en
lecturas subsecuentes o escrito del archivo.

Este nombre interno es actualmente un puntero, llamado un "archivo puntero", para una estructura que contiene informacion acerca del archivo, tal como la ubicacion de un buffer, la posicion del caracter en curso en el buffer, si el archivo esta siendo leido o escrito, y algo semejante. Los usuarios no necesitan conocer lo detalles, porque parte de las definiciones de I/O standard obtenidas desde stdio.h es una definicion de estructura llamada FILE. La sola declaracion necesitada por un archivo puntero esta ejemplificada en

```
FILE *fopen(), *fp;
```

Esto dice que fp es un puntero para un FILE, y fopen devuelve un puntero
para un FILE. Note que FILE es un tipo de nombre, como int, no un rotulo de estructura; esto es implementado como un typedef. (Detalles de como trabaja todo esto en el sistema UNIX esta dado en el cap.8.)

La actual llamada para fopen en un programa es

```
fp = fopen (name, mode);
```

El primer argumento de fopen es el name del archivo, como un caracter de
string. El segundo argumento es mode, tambien como caracter de string,
que indica como uno intenta usar el archivo. Modos permisibles son leer ("r"), escribir ("w"), o agregar ("a").

Si Ud. abre un archivo que no existe, para escribir o agregar, es creado (si es posible). Abriendo un archivo existente para escribir, provoca que el antiguo contenido sea descartado. Tratar de leer un archivo que no existe es un error, y alli pueden estar otras causas de error (como tratar de leer un archivo cuando Ud. no tiene permiso). Si hay algun error, fopen devolvera el valor del puntero nulo que es el valor NULL (que por conveniencia esta tambien definido en stdio.h).

Lo proximo que necesitamos es una manera para leer o escribir el archivo una vez que esta abierto. Hay varias posibilidades, de las cuales getc y putc son las mas simples. getc devuelve el proximo caracter desde un archivo; esto necesita un archivo puntero para indicarle que archivo. Asi

```
c = getc (fp)
```

ubica en c el proximo caracter desde el archivo referido por fp, y EOF cuando encuentra el fin de archivo.

putc es lo inverso de getc:

```
putc (c,fp)
```

pone el caracter c en el archivo fp y devuelve c. Como getchar y putchar, getc y putc pueden ser macros en vez de funciones.

Cuando un programa es puesto en marcha, tres archivos son abiertos automaticamente, y archivos punteros son provistos de ellos. Estos archivos son la entrada standard, la salida standard y la salida de error standard; los correspondientes archivos punteros son llamados stdin, stdout, y stderr. Normalmente, estos son conectados al terminal, pero stdin y stdout pueden ser re-dirigidos a archivos o pitos, como describimos en la seccion 7.2.

getchar y putchar pueden ser definidos en terminos de getc, putc, stdin y stdout como sigue:

```
#define   getchar()     getc(stdin)
#define   putchar()     putc(c, stdout)
```

Para el formatesdo de los archivos de entrada o salida, las funcionesfscanf y fprintf pueden ser usadas. Estas son identicas a scanf y printf, salvo que el primer argumento es un archivo puntero que especifica el archivo a ser leido o escrito; el string de control es el segundo argumento.

Con estos preparativos, ahora estamos en posicion para escribir el programa cat para concatenar archivos. El diseno basico es uno que ha sido encontrado conveniente para muchos programas; si hay argumentos de
linea de comando, ellos son procesados en orden. Si no hay argumentos, la entrada standard es procesada. De esta forma el programa puede ser usado en una sola posicion o como parte de un gran proceso.

```
#include <stdio.h>

main(argc, argv)    /* cat: concatena archivos */
int argc;
char *argv [];
{
    FILE *fp, fopen();

    if (argc == 1)  /* no args; copia entrada standard
                       */
        filecopy(stdin);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL)
            {
                    printf("cat: no puede abrirlo
                     %\n", *argv);
                    break;

                }
            else {
                filecopy(fp);
                fclose(fp);
                }
}

filecopy (fp)  /* copia archivo fp a salida standard
                  */
FILE *fp;
{
    int c;

    while ((c = getc(fp)) != EOF)
        putc(c, stdout);
}
```

Los archivos punteros stdin y stdout son pre-definidos en la biblioteca de I/O como en la entrada y salida standard; ellas pueden usadas en cualquier sitio; puede estar un objeto del tipo FILE *. Ellas son constantes, sin embargo, "no" variables, asi no trata de asignarselas.

La funcion fclose es lo inverso de fopen; interrumpe la coneccion entre el archivo puntero y el nombre externo que fue establecido por fopen, liberando el archivo puntero para otro archivo. Ya que la mayor parte de los sistemas operativos tienen algunos limites en el numero de simultaneidad para abrir archivos que un programa puede tener, es una buena idea liberar cosas cuando ellas no son necesarias, como hicimos en
cat. Hay tambien otra razon para fclose en un archivo de salida - esto fluye al buffer en que putc esta recogiendo salida. (fclose es llamado automaticamente por cada archivo abierto cuando un programa termina normalmente.)

7.7 MANEJO DE ERRORES - STDERR Y EXIT

El tratamiento de errores en cat no es ideal. El problema es que si uno de los archivos no puede ser accesado por alguna razon, el diagnostico es impreso al final de la salida concatenada. Esto es aceptable si esta salida va al terminal, pero esta mal si va dentro de un archivo o en otro programa via un pipeline.

Para manejar esta situacion mejor, un segundo archivo de salida, llamado stderr, es asignado a un programa en la misma forma que estan stdin y stdout. Si todo es posible, la salida escrita en stderr aparece en el terminal del usuario aun si la salida standard es redirigida.

Revisemos cat para escribir sus ensajes de error en el archivo de error standard.

```
#include <stdio.h>

main (argc, argv)  /* cat: concatena archivos  */
int argc;
char *argv [];
{
    FILE *fp, *fopen();

    if(argc == 1)  /* no args; copia entrada standard
                      */
        filecopy(stdin);
    else
        while (--argc > 0)
            if((fp = fopen(*argv, "r")) == NULL) {
                fprintf(stderr, "cat: no puede
                 abrirlo %s\n", *argv);
                exit(10;
            } else  {
                filecopy(fp);
                fclose(fp);
            }
    exit(0);
}
```

El programa indica dos formas de error. El diagnostico de salida producido por fprintf va dentro de stderr, asi encuentra su forma para el terminal del usuario en vez de desaparecer bajo un pipeline o dentro de un archivo de salida.

El programa tambien usa la funcion de biblioteca exit, que termina la ejecucion del programa cuando es llamada. El argumento de exit esta
disponible para cualquier proceso llamado, asi el exito o fracaso de un programa puede ser evaluadao por otro programa que use esto como un subproceso. Por convencion, si devielve un cero esta todo bien, y varios valores no-ceros indican situaciones anormales.

exit llama a fclose para cada archivo de salida, para fluir fuera

de un buffer de salida, luego llama a una rutina llamada _exit. La funcion _exit provoca inmediatamente el termino sin fluir a algun buffer; esto puede ser llamado directamente, si lo desea.

7.8 LINEA DE ENTRADA Y SALIDA

La biblioteca standard proporciona una rutina fgets que es completamente similar a getline. La llamada

```
fgets(line, MAXLINE, fp)
```

lee la proxima linea de entrada (incluyendo el newline) desde el archivo fp en el arreglo de caracter line;a lo mas MAXLINE-1 caracteres seran leidos. La linea resultante es terminada con \0. Normalmente, fgets devuelve line; en el fin de archivo este devuelve NULL. (Nuestro getline devuelve el largo de la linea, y cero para fin de archivo.)

Para la salida,la funcion fputs escribe un string (que no necesita contener un newline) para un archivo:

```
fputs (line, fp)
```

Para mostrar que no hay magia acerca de funciones como fgets y fputs, aqui estan, copiadas directamente de la biblioteca standard de I/O:

```
#include <stdio.h>

char *fgets(s, n, iop)  /* obtiene alo mas n caracteres
                   desde iop  */
char *s;
int n;
register FILE *iop;
{
    register int c;
    register char *cs;

    cs = s;
    while (--n > 0 && (c = getc(iop)) != EOF)
        if ((*cs++ = '\n')
            break;
    *cs = '\0';
    return((c == EOF && cs == s) ? NULL : s);
}
```

```
fputs(s, iop)   /* pone el string s en el archivo iop
                           */
register char *s;
register FILE *iop;
{
    register int c;

    while (c = *s++)
        putc(c, iop)
}
```

7.9 MISCELANEA DE ALGUNAS FUNCIONES

La biblioteca standard proporciona una variedad de funciones, unas pocas tan especiales como utiles. Ya hemos mencionado las funciones de
string: strlen, strcpy, strcat y strcmp. Aqui hay algunas otras.

ENSAYANDO CLASES Y CONVERSIONES DE CARACTERES

Varias macros ejecutan preguntas de caracter y conversiones:

```
isalpha(c)    no-cero si c es alfabetica, 0 si no.
isupper(c)    no-cero si c es minuscula, 0 si no.
islower(c)    no-cero si c es mayuscula, 0 si no.
isdigit(c)    no-cero si c es digito, 0 si no.
isspace(c)    no-cero si c es blanco, tab o newline, 0
        si no.
toupper(c)    convierte c a mayuscula.
tolower(c)    convierte c a minuscula.
```

UNGETC

La biblioteca standard proporciona una version mas bien restringida de la funcion ungetc que escribimos en el cap.4; es llamada ungetc.

```
ungetc(c, fp)
```

pone el caracter c de vuelta en el archivo fp. Solo un caracter de pushback (lo empuja de vuelta) es permitido por archivo. ungetc puede ser usada con algunas de las funciones de entrada y macros como scanf, getc, o getchar.

SISTEMA DE LLAMADA

La funcion system(s) ejecuta el comando contenido en el caracter string s, luego resume la ejecucion del programa en curso. El contenido de s depende fuertemente del sistema operativo local. Como un ejemplo trivial, en UNIX, la linea

```
system("date");
```

causa que el programa date sea corrido; este imprime la fecha y hora del dia.

MANEJO DEL ALMACENAJE

La funcion calloc es mas bien como la alloc que usamos en **CAPITULO**s previos.

```
calloc(n, sizeof(objeto))
```

devuelve un puntero para bastante espacio para pbn objetos del largo especificado, o NULL si el requerimiento no puede ser satisfecho. El almacenaje esta inicializado en cero.

El puntero tiene el alineamiento propio para el objeto en cuestion, pero esto seria lanzado en el tipo apropiado, como en

```
char *calloc();
int *ip;

ip = (int *) calloc(n, sizeof(int));
```

cfree(p) libera el espacio apuntado por p, donde p es originalmente obtenido por un llamado a calloc.

El **CAPITULO** 8 muestra la implementacion de un asignador de almacenaje como calloc, en que los blocks asignados pueden ser liberados
en algun orden.

_