

En este curso usaremos:

- **Gnu/Linux** más el **Compilador de lenguaje C de GNU (gcc)**
- **FLEX: Generador de Analizadores Léxicos Rápidos** (para **reconocimiento de patrones**)
- **BISON: Generador de Analizadores Sintácticos** (para la **comprensión del significado de los patrones, la GRAMÁTICA**)

Un poco de Historia:

- **1973: Dennis Ritchie** crea el **Lenguaje de Programacion C**
- {Universidades de EE.UU + EMPRESAS (General Electric, etc) + Ken Thompson} crean un S.O. denominado **MULTICS** (“**MULTIPLEXER INFORMATION CONTROL SYSTEMS**”)
- **Ken Thompson** toma una computadora denominada **DEC P.D.P 7** (de la EMPRESA DEC, “**DIGITAL EQUIPMENT CORPORATION**”) para hacer un S.O. que le permita poder jugar al “**SPACE INVADERS**” ==> crea el “**UNICS**”, este es el **primer sistema operativo que no se implementa en LENGUAJE ASSEMBLER**, pues fue implementado totalmente en **LENGUAJE C**. Es un S.O. “monousuario”
- **1978: The C Programming Languaje – Kernighan & Ritchie**
- **1986: Se crea el primer comité de estandarización de lenguajes de programación “X3J11” (QUE LUEGO SE LLAMARÁ ANSI: AMERICAN NATIONAL STANDAR INSTITUTE)**
- **1989: se crea el ANSI C**
- **1990: se crea el ISO C**
- **1999: se crea el ISO C '99 (“C99”)**

C tiene los siguientes tipos de datos:

- **chart**
- **short**
- **int**
- **long**
- **(gcc) long long**

Se estableció que se cumple la siguiente regla:

$$\text{sizeof(char)} \leq \text{sizeof(short)} \leq \text{sizeof(int)} \leq \text{sizeof(long)}$$

Nota: si la Arquitectura del Microprocesador tiene como tamaño de palabra, 32 bits, por lo general se deja que el tipo de dato int ocupe en memoria 32 bits para que sea más facil de realizar las lecturas y escrituras en el bus de datos.

Números Binarios:

Supongamos números de 8 bits, que lo representaremos simbólicamente de la siguiente manera

$$b_7 \ b_6 \ b_5 \ b_4 \ b_3 \ b_2 \ b_1 \ b_0$$

donde cada b_i puede tomar solo el valor CERO o el valor UNO
(base 2, porque sólo usa 2 símbolos para representar los números) y cada b_i tiene un peso diferente relacionado a su posición en el número. Observar que las posiciones se enumeran desde la cifra menos significativa, del extremo derecho, comenzando con b_0 , hacia la cifra mas significativa del extremo izquierdo, en este caso b_7 .

Entonces, cualquier número decimal (en base 10, por ejemplo: 12) se puede descomponer en la siguiente suma polinómica:

$$N_{10} = b_7 * 2^7 + b_6 * 2^6 + b_5 * 2^5 + b_4 * 2^4 + b_3 * 2^3 + b_2 * 2^2 + b_1 * 2^1 + b_0 * 2^0$$

Por ejemplo:

$$12 = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0 = 1*8 + 1*4 + 0*2 + 0*1 = 8 + 4$$

La técnica para pasar desde una representación de un número en el sistema decimal hacia el sistema binario es la que se muestra en el siguiente ejemplo:

$$12 = (?) \text{ "en base 2"}$$

$$\begin{array}{r} 12 \\ \underline{| 2} \\ 0 \quad 6 \quad \underline{| 2} \\ \quad 0 \quad 3 \quad \underline{| 2} \\ \quad \quad 1 \quad \underline{| 2} \\ \quad \quad \quad 1 \quad 0 \end{array}$$

Se toman los restos desde abajo hacia arriba
Obteniéndose la siguiente equivalencia de
representaciones de números
 $12_{\text{base } 10} = 1100_{\text{base } 2} = 1*2^3 + 1*2^2$

Siguiendo con este esquema de 8 dígitos binarios (8 cifras que pueden tomar el valor cero o el valor uno) obtenemos un **unsigned char** (**un carácter sin signo**)

¿Cómo obtenemos la representación de números con signos?

Las posibles soluciones son:

- a) usando el signo menos “-” : NO SE PUEDE, pues el sistema de representación es binario, es decir los únicos símbolos que admite son “0” o “1”
- b) Utilizar uno de los bits (bits significa dígito binario) como el **bit de signo**, por ejemplo, usar el bit del extremo izquierdo como indicador del signo tomando como convención:
 - > Si es un CERO ==> el número es POSITIVO
 - > Si es un UNO ==> el número es NEGATIVO

Pero si observamos bien, al utilizar esta representación para los números binarios encontramos el siguiente problema: **TENEMOS DOS REPRESENTACIONES PARA UN MISMO NÚMERO, ES DECIR**

$$(+ 0)_{\text{BASE } 10} = \textcolor{red}{00000000}_{\text{BASE } 2}$$

$$(- 0)_{\text{BASE } 10} = \textcolor{red}{10000000}_{\text{BASE } 2}$$

- c) Utilizamos el sistema de representación **EN COMPLEMENTO A DOS**, el cual requiere de tres sencillas acciones:
 - 1º) Obtener la representación en binario del valor absoluto (el valor sin considerar el signo)
Por ejemplo, vimos que para 12 es 1100
 - 2º) Obtener el **COMPLEMENTO A UNO** de esa representación binaria obtenida, es decir, donde hay un cero se lo intercambia por un uno y viceversa
Para el mismo ejemplo sería:
12 EN DECIMAL <=> 1100 EN BINARIO (VALORABSOLUTO)
12 EN DECIMAL <=> 0011 EN BINARIO (COMPLEMENTO A UNO)

3º) Sumar “1” al COMPLEMENTO A UNO para obtener finalmente el valor que está representado en **COMPLEMENTO A DOS**

$$\begin{array}{rcl} 12 \text{ EN DECIMAL} & \Leftrightarrow & 0011 \text{ EN BINARIO (COMPLEMENTO A UNO)} \\ & & \underline{\pm 1} \\ & & 0100 \text{ EN BINARIO (COMPLEMENTO A DOS)} \end{array}$$

NOTA: Las sumas y restas en binario a se realizan de la siguiente manera:

$$\begin{array}{l} 1 + 1 = 10 \quad (\text{es decir "se lleva uno"}) \\ 10 - 1 = 1 \end{array}$$

De esta manera se logra una representación de números en binario en la cuál el cero no posee una doble representación.

El máximo número representable es: $0111111_2 = 127_{10}$

El mínimo número representable es: $1000000_2 = -128_{10}$

Primer programa en C:

Con cualquier editor de texto se confexiona el siguiente archivo que podemos nombrarlo:

“holamundo.c”. Por convención todos los **archivos que sean código fuente escrito en Lenguaje C llevan el sufijo “.c”**. A continuación se muetra el contenido del primer código fuente de un programa escrito en C, por nosotros:

```
// Lo que le sigue a las dos barras es un comentario, el compilador ignora esta línea
// Los comentarios son de extrema importancia para explicitar la documentación del código.
```

```
#include <stdio.h> // Aquí se le dice al compilador de C que incluya el archivo
                    // que en general se encuentra en /usr/include/stdio.h
                    // Se trata de un archivo que define funciones en C de La Librería de
                    // Entrada y Salida Estandar ( stdio : Standar Input Output)
                    // Estos archivos se los denominan HEADERS o ARCHIVOS DE
                    // ENCABEZADO y por convención terminan con el sufijo “.h”

int main()          // Todo programa posee una función main, indica que de aquí en adelante
{                  // comienza el código ejecutable que se traducirá en órdenes para el
                // microprocesador. El “int” que aparece significa que el programa
                // retornará al ambiente de ejecución un valor “entero”
                // Es convención en UNIX y GNU/LINUX:
                // retornar 0 si la operación se ha realizado satisfactoriamente
                // retornar 1 si la operación no se concretó satisfactoriamente

    printf("Hola Mundo! \n"); // Se hace uso de una función de stdio.h que se encarga de
                            // mostrar en pantalla el mensaje “Hola Mundo!” y
                            // procede a realizar un salto de línea y retorno al inicio
    return 0;
}

// Fin del archivo “holamundo.c”
```

Hasta aquí solo hemos editado texto, solo nos falta realizar el proceso de compilación del código fuente. Para ello emplearemos el **Compilador de C de GNU (GCC)** invocándolo desde la línea de comandos de la siguiente manera:

gcc holamundo.c -o holamundo.out

De este modo si no hemos cometido errores de sintaxis o de lógica, se generará el fichero ejecutable “holamundo.out” (Por convención se usa el sufijo “.out” pero no es obligatorio). Si hay errores se informarán en pantalla indicando en qué línea del archivo de código fuente está el error.

Operaciones con bits:

A nivel de bits (dígitos binarios, 0 o 1) nos encontraremos con las siguientes operaciones:

- PRODUCTO LÓGICO: cuyo operador es & (AND)
- SUMA LÓGICA: cuyo operador es | (OR)
- SUMA LÓGICA EXCLUSIVA: cuyo operador es ^ (OR EXCLUSIVE)
- COMPLEMENTO: cuyo operador es ~

Veremos algunos ejemplos en la siguiente tabla:

OPERACIONES CON BITS						
A	B	A & B	A B	A ^ B	~ A	~ B
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	1
1	1	1	1	0	0	0

También hay operaciones para la **manipulación de números binarios**, denominadas

OPERACIONES DE DESPLAZAMIENTO A NIVEL DE BITS:

Suponiendo que a es un número binario, por ejemplo **a = 10010001** (en C sería un dato del tipo **char**)

- **Desplazamiento de “n” bits hacia la derecha: se simboliza “>> n”**

Por ejemplo, para el mismo a podríamos tener, **a >> 3**, lo que daría como resultado la siguiente secuencia de bits:

a >> 3 es equivalente a **00010010** Es decir corre 3 posiciones a la derecha y lo que queda libre se rellena con ceros (en rojo)

- **Desplazamiento de “n” bits hacia la izquierda: se simboliza “<< n”**

Por ejemplo, para el mismo a podríamos tener, **a << 3**, lo que daría como resultado la siguiente secuencia de bits:

a << 3 es equivalente a **10001000** Es decir corre 3 posiciones a la izquierda y lo que queda libre se rellena con ceros (en rojo)

Nota: En general se cumple:

a << k ≡ a * 2^k Desplazar k bits hacia la izquierda es equivalente a multiplicar por 2^k

Ahora veamos...

¿Cómo multiplicar dos números enteros sin signos cuando suponemos que no tenemos una operación de multiplicación definida en nuestro compilador de C?

Supongamos que poseemos:

- suma
- desplazamiento hacia la izquierda y hacia la derecha
- operaciones con bits

Este problema está resuelto en unos papiros que encontró **Rhind** en el año 1858, los cuales están codificados en lenguaje hiératico (forma cursiva del jeroglífico) que contenía lecciones de Aritmética y fue compuesto por un escriba llamado Ahmés.

Supongamos que **m** y **n** son dos números enteros tales que: $m > 0$ y $n > 0$, entonces podemos definir al producto de la siguiente manera:

$$\begin{aligned} \text{prod}(m, n) = & \begin{cases} m, & \text{si } n = 1 \\ \text{prod}(2*m, n/2), & \text{si } n \text{ es par} \\ \text{prod}(2*m, (n - 1)/2) + m, & \text{si } n \text{ es impar} \end{cases} \end{aligned}$$

Demostración:

$$\begin{aligned} \text{Si } n \text{ es par} \implies & n = 2*k \text{ para todo } k \text{ entero distinto de cero} \implies \\ \implies & m*n = m*2*k = 2*m*k \implies \text{como } k = n/2 \implies \\ \implies & m*n = 2*m*k = (2*m)*(n/2) = \text{prod}(2*m, n/2) \end{aligned}$$

$$\begin{aligned} \text{Si } n \text{ es impar} \implies & n = 2*k + 1 \text{ para todo } k \text{ entero distinto de cero} \implies \\ \implies & m*n = m*(2*k + 1) = 2*m*k + m \implies \text{como } k = (n - 1)/2 \implies \\ \implies & m*n = 2*m*k + m = 2*m*[(n - 1)/2] + m = \text{prod}(2*m, (n - 1)/2) + m \end{aligned}$$

Vamos a pasar esta función matemática a código fuente en Lenguaje C. Recordemos los supuestos hechos (contamos con un compilador que solo posee suma, desplazamientos de bits hacia la izquierda y hacia la derecha, junto con operaciones con bits) entonces resulta evidente que usaremos las siguientes relaciones:

$2*m \equiv m << 1$ Multiplicar **m** por 2 equivale a desplazar 1 bit hacia la izquierda a **m**

$n/2 \equiv n >> 1$ Dividir **n** por 2 equivale a desplazar 1 bit hacia la derecha a **m**

n es par Si el bit menos significativo (el del extremo derecho) de **n** es cero

n es impar Si el bit menos significativo (el del extremo derecho) de **n** es uno

Veamos el siguiente trozo de código en C:

```
if( n & 1 == 0 )      // Si( n AND 1 es igual a 0 )
{
    // entonces mostrar "n es par!"
    printf( "n es par! \n");
    // pues n debe poseer el bit menos significativo en cero
}
// para que la comparación de CERO
else if( n & 1 == 1 ) // SINÓ, SI ( n AND 1 es igual a 1 )
{
    // entonces mostrar "n es impar!"
    printf(" n es impar! \n");
    // pues n debe poseer el bit menos significativo en uno
}
// para que la comparación de UNO
```

Entonces una primera aproximación al código fuente en Lenguaje C buscado puede ser la siguiente:

```
/* Aquí comienza comentario que ocupa mas de una
* línea. Esto es ignorado por el compilador */
```

// Archivo: productoUno.c

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

```
/* A continuación se especifica el prototipo de la función prod . Es buenas prácticas de programación
* incluir siempre al principio del código fuente los prototipos de las funciones que luego van a ser
* usadas. Estos indican el tipo de dato que retorna la función, seguido de el nombre de la función,
* y entre parentesis, los tipos de datos que la función toma como parámetros, separados por
* coma. El prototipo de la función debe terminar con punto y coma.
```

```
*/
```

```
unsigned int prod( unsigned int , unsigned int );
```

```
// Ahora se realiza la implementación del código en C de la función
```

```
unsigned int prod( unsigned int m , unsigned int n )
```

```
{
```

```
    if( n == 1 )
        return m;
    else    if( (n & 1) == 0 )           // Si n es par
            return prod( m << 1 , n >> 1 );
    else    if( (n & 1) == 1 )           // Si n es impar
            return ( prod( m << 1 , (n - 1) >> 1 ) + m );
```

```
}
```

```
int main()      // comienza el código ejecutable
```

```
{      int a = 2;
```

```
    int b = 3;
```

```
    printf( "a vale: %d \n" , a );
```

```
    printf( "b vale: %d \n" , b );
```

```
    int producto = prod( a , b );
```

```
    printf( "el producto a*b vale: %d \n" , producto );
```

```
    int opcion = 0;
```

```
    while( opcion == 0 ) // Bucle de repetición: MIENTRAS( opcion es igual a 0 )
```

```
    {                      // hacer
```

```
        printf( "Introduzca un nuevo valor para a: " ); // imprime en pantalla el mensaje
```

```
        scanf("%d" , &a ); // Lee desde teclado un decimal (%d) y lo guarda en la variable a
```

```
        printf( "Introduzca un nuevo valor para b: " );
```

```
        scanf( "%d" , &b ); // Lee desde teclado un decimal (%d) y lo guarda en la variable b
```

```
        producto = prod( a , b );
```

```
        printf( "el producto a*b vale: %d \n" , producto );
```

```
        printf( "Desea ingresar otros valores para a y b ? \n" );
```

```
        printf( "    Ingrese 0 para dar nuevos valores! \n" );
```

```
        printf( "    Ingrese un valor distinto de cero para finalizar! \n" );
```

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

```
}
```

```
return 0;      // indica que se ha realizado todo con éxito al ambiente donde se ejecuta
```

```
}
```

```
// Fin del Archivo: productoUno.c
```

¿Se puede mejorar el código fuente de este programa?

Pues, sí.

Para ello veremos una técnica denominada **RECUSIÓN DE COLA (“TAIL RECURSION”)**.

Una **función es recursiva de cola** (“tr” de tail recursion) si cumple los siguientes postulados:

- a) Devuelve constantes o expresiones simples
- b) Cuando se llama a sí misma retorna sin ninguna otra operación

Ejemplo:

La siguiente **función matemática** denominada “**factorial de un número n**” se define de la siguiente manera:

$$\begin{array}{l} \text{fact(} n \text{) = } \\ | \quad \text{INDEFINIDO , si } n < 0 \\ | \\ | \quad \text{1 , si } n = 0 \\ | \\ | \quad \text{fact(} n - 1 \text{) , si } n > 0 \\ | \end{array}$$

La implementación de esta función en Lenguaje C se detalla a continuación (en la siguiente hoja):

// Archivo: factorial.c

```
// prototipo de la función factorial  
int factorial( unsigned int );
```

// Implementación de la función factorial en Lenguaje C

```
int factorial( unsigned int n )
```

{

```
if( n == 0 ){
    return 1;
}
else if( n > 0 ){
    return ( n * factorial( n - 1 ) );
}
else if( n < 0 ){
    printf("\n ERROR! \n");
    return 0;
}
```

// Comienza el código ejecutable

```
int main()
```

{

```
int a=0, fact=1;
```

```
int opcion = 0;
```

```
while( opcion == 0 ) // Bucle de repetición: MIENTRAS( opcion es igual a 0 )
```

{ // hacer

```
printf( "Introduzca un valor entero positivo para a: " );
```

```
scanf("%d", &a); // Lee desde teclado un decimal (%d) y lo guarda en a
```

```
if( a >= 0 ){
```

```
fact ≡ factorial( a );
```

}

}
else{

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

}

1
n

```
primario. El factorial de a vale: a! = %d \n", fact );
printf( "Desea ingresar otro valor para a? \n" );
```

```
printf( "Desea ingresar otro valor para a: %f\n",  
printf( "    Ingrese 0 para dar nuevo valor! \n'"
```

Ingrese 0 para dar nuevo valor: **0**,
Ingresar un valor distinto de cero para

primu("Ingresé un valor distinto de cero para finalizar: "),
scanf("%d" &opcion);

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

۱

11

// Fin del archivo: factorial.c

Como vemos esta función no cumple con los postulados, por lo tanto no es recursiva de cola. Pero podemos realizar una implementación de esta función que sea recursiva de cola.

¿Se puede hacer a esta función recursiva de cola?

Sí, pues veamos el siguiente trozo de código fuente en Lenguaje C:

```
int factorialtr( unsigned int n , unsigned int resultado )
{
    if( n == 0 ) return resultado;           // resultado deberá valer 1 para que funcione
    else    if( n > 0 ){
        return factorialtr( (n - 1) , resultado*n );
    }
    else    if( n < 0 ){
        printf("\n ERROR!!! \n");
        return 0;
    }
}
```

Y notamos que: **factorial(n) <==> factorialtr(n , 1)**

A continuación detallamos el código fuente en Lenguaje C de un programa que hace uso de la función recién definida factorialtr (en la próxima hoja):

```
// Archivo: factorialtr.c

// prototipo de la función factorialtr
int factorialtr( unsigned int , unsigned int );

// Implementación de la función factorial en Lenguaje C
int factorialtr( unsigned int n , unsigned int resultado )
{
    // resultado=1; para que sea como el factorial
    if( n == 0 ){
        return resultado;
    }
    else  if( n > 0 ){
        return factorialtr( (n - 1) , (n * resultado) );
    }
    else  if( n < 0 ){
        printf("ERROR!");
        return 0;
    }
}

// Comienza el código ejecutable
int main()
{
    int a=0, facttr=1;
    int opcion = 0;
    while( opcion == 0 ) // Bucle de repetición: MIENTRAS( opcion es igual a 0 )
    {
        // hacer
        printf( "Introduzca un valor entero positivo para a: " );
        scanf("%d" , &a ); // Lee desde teclado un decimal (%d) y lo guarda en a
        if( a >= 0 ){
            facttr = factorialtr( a , 1 );
        }
        else{
            printf("\neeeeeeeeeeeeERRORrrrrrrrrrrr!\n");
            return 1; // salida abrupta del programa por condición errónea
        }
        printf( "El factorial de a vale: a! = %d \n" , facttr );
        printf( "Desea ingresar otro valor para a? \n" );
        printf( "    Ingrese 0 para dar un nuevo valor! \n" );
        printf( "    Ingrese un valor distinto de cero para finalizar! \n" );
        scanf( "%d" , &opcion );
    }
    return 0; // salida exitosa del programa todo ha salido bien
}
// Fin del archivo: factorialtr.c
```

Una **característica interesante de las funciones recursivas de cola** es que **al ejecutarlas ocupan una cantidad de memoria que permanece constante en el tiempo** sin variar durante la ejecución del programa. Una función recursiva de cola se comporta como una iteración.

Ahora transformaremos esta función, que implementa al factorial, en una iteración:

```
// Archivo: factorialiteracion.c
// prototipo de la función factorialtr
int factorialiteracion( unsigned int , unsigned int );
// Implementación de la función factorial, como una iteración, en Lenguaje C
int factorialiteracion( unsigned int n , unsigned int resultado )
{
    // resultado=1; para que sea como el factorial
    sigue: // define una etiqueta que sirve como referencia para usar con “goto”
        // Se dice que usar la sentencia “goto” junto con etiquetas NO ES UNA
        // BUENA TÉCNICA DE PROGRAMACIÓN. Los programas que hizo Bill
        // Gates están llenos de sentencias goto con etiquetas

    if( n == 0 ){
        return resultado;
    }
    else if( n > 0 ){
        resultado = (n * resultado) ;
        n = (n - 1);
        goto sigue; // así se le dice “ve hasta la dirección en memoria etiquetada con
    } // “sigue”
    else if( n < 0 ){
        printf("ERROR!");
        return 0;
    }
}
// Comienza el código ejecutable
int main()
{
    int a=0, factiter=1;
    int opcion = 0;
    while( opcion == 0 ) // Bucle de repetición: MIENTRAS( opcion es igual a 0 )
    {
        // hacer
        printf( "Introduzca un valor entero positivo para a: " );
        scanf("%d" , &a ); // Lee desde teclado un decimal (%d) y lo guarda en a
        if( a >= 0 ){
            factiter = factorialiteracion( a , 1 );
        }
        else{
            printf("\neeeeeeeeeeeeERRORrrrrrrrrrrrr!\n");
            return 1; // salida abrupta del programa por condición errónea
        }
        printf( "El factorial de a vale: a! = %d \n" , factiter );
        printf( "Desea ingresar otro valor para a? \n" );
        printf( "    Ingrese 0 para dar un nuevo valor! \n" );
        printf( "    Ingrese un valor distinto de cero para finalizar! \n" );
        scanf( "%d" , &opcion );
    }
    return 0; // salida exitosa del programa todo ha salido bien
}
// Fin del archivo: factorialtr.c
```

Pero se dice que no es bueno usar el “goto”. Tenemos que sacar el “goto” y la “etiqueta”!!!

```
// Archivo: factorialbucle.c
// prototipo de la función factorialbucle
int factorialbucle( unsigned int );

// Implementación de la función factorial utilizando un bucle de repetición while
// (“MIENTRAS”) en Lenguaje C
int factorialbucle( unsigned int n )
{
    int resultado = 1; // declara una variable local, visible y utilizable solo dentro de esta
                      // función
    while( n > 0 )
    {
        resultado *= n;      // equivale a: resultado = resultado * n
        n--;                // equivale a: n = n - 1
    }
    if( n >= 0 ){
        return resultado;
    }
    else if( n < 0 ){
        printf("\neeeeeeeeeeeeERROR!rrrrrrrrrrrr\n");
        return 0;
    }
}

// Comienza el código ejecutable
int main()
{
    int a=0, factbucle=1;
    int opcion = 0;
    while( opcion == 0 )    // Bucle de repetición: MIENTRAS( opcion es igual a 0 )
    {
        // hacer
        printf( "Introduzca un valor entero positivo para a: " );
        scanf( "%d" , &a );    // Lee desde teclado un decimal (%d) y lo guarda en a
        if( a >= 0 ){
            factbucle = factorialbucle( a );
        }
        else{
            printf("\neeeeeeeeeeeeERRORrrrrrrrrrrrr!\n");
            return 1;    // salida abrupta del programa por condición errónea
        }
        printf( "El factorial de a vale: a! = %d \n" , factbucle );
        printf( "Desea ingresar otro valor para a? \n" );
        printf( "    Ingrese 0 para dar un nuevo valor! \n" );
        printf( "    Ingrese un valor distinto de cero para finalizar! \n" );
        scanf( "%d" , &opcion );
    }
    return 0;    // salida exitosa del programa todo ha salido bien
}
// Fin del archivo: factorialbucle.c
```

Continuando con el ejemplo de la **función producto**, ahora vamos a realizar su **implementación para que sea recursiva de cola**.

// Archivo: productotr.c

```
#include <stdio.h>

// A continuación se especifica el prototipo de la función prodr
unsigned int prodr( unsigned int , unsigned int , unsigned int );

// Ahora se realiza la implementación del código en C de la función prodr
unsigned int prodr( unsigned int m , unsigned int n , unsigned int resultado )
{
    if( n == 1 ){
        return (resultado + m);           // deberá ser resultado = 0 para que sea como el
    }                                   // producto
    else    if( (n & 1) == 0 ){          // Si n es par
        return prodr( m << 1 , n >> 1 , resultado );
    }
    else    if( (n & 1) == 1){          // Si n es impar
        return ( prodr( m << 1 , ( (n - 1) >> 1 ) , resultado + m ) );
    }
}

int main()      // comienza el código ejecutable
{
    int a = 2;
    int b = 3;
    int res = 0;    // debe valer cero para que sea el producto lo que se calcula
    printf( "a vale: %d \n" , a);
    printf( "b vale: %d \n" , b );
    int producto = prodr( a , b , res );
    printf( "el producto a*b vale: %d \n" , producto );
    int opcion = 0;
    while( opcion == 0 ) // Bucle de repetición: MIENTRAS( opcion es igual a 0 )
    {
        // hacer
        printf( "Introduzca un nuevo valor para a: " ); // imprime en pantalla el mensaje
        scanf("%d" , &a );// Lee desde teclado un decimal (%d) y lo guarda en la variable a
        printf( "Introduzca un nuevo valor para b: " );
        scanf( "%d" , &b ); // Lee desde teclado un decimal (%d) y lo guarda en la variable b
        int resx = 0;
        int productox = prodr( a , b , resx );
        printf( "el producto a*b vale: %d \n" , productox );
        printf( "\nDesea ingresar otros valores para a y b ? \n" );
        printf( "    Ingrese 0 (CERO) para dar nuevos valores! \n" );
        printf( "    Ingrese un valor distinto de cero para finalizar! \n" );
        scanf( "%d" , &opcion );
    }
    return 0;                  // indica que se ha realizado todo con éxito al ambiente donde se ejecuta
}

// Fin del Archivo: productotr.c
```

Hay que notar que al realizar la invocación a la función: `prodtr(a , b , res)`; el valor que posee la variable entera `res` es **CERO**, pues de otra manera esta función no devolvería como resultado el producto entre dos números enteros positivos.

Ahora pasaremos a ver un tema muy importante que todo buen programador que use el Lenguaje C debería dominar sin problemas: (este tema ha sido calificado como merecedor de “CUATRO CALABERAS” por el mismo profesor Guido Macchi, debido a la complejidad que supone para los alumnos que lo ven por primera vez).

PUNTEROS EN C:

En C una variable tiene:

- a) un **tipo de dato**
- b) un **valor** (correspondiente a ese tipo de dato)
- c) una **dirección de memoria**.

Para lograr comprender esto debemos recordar el concepto que el mismísimo **Jon Von Neuman** acuñó en unos **informes preliminares de los años 1940**, él hablaba de una **máquina que tenía la capacidad de almacenar datos y programas en memoria** (el **concepto de programa almacenado en memoria**). Así en la memoria de una computadora hay **datos e instrucciones que manipulan esos datos** y ambos, **datos e instrucciones están identificados**, para facilitar su manipulación, mediante lo que denominamos las **direcciones de memoria**. La unidad mínima direccionable de memoria depende de la Arquitectura de la Computadora. Por lo general las computadoras permiten direccionar hasta el nivel de bytes (1 byte = 1 octeto = 8 bits = 8 dígitos binarios).

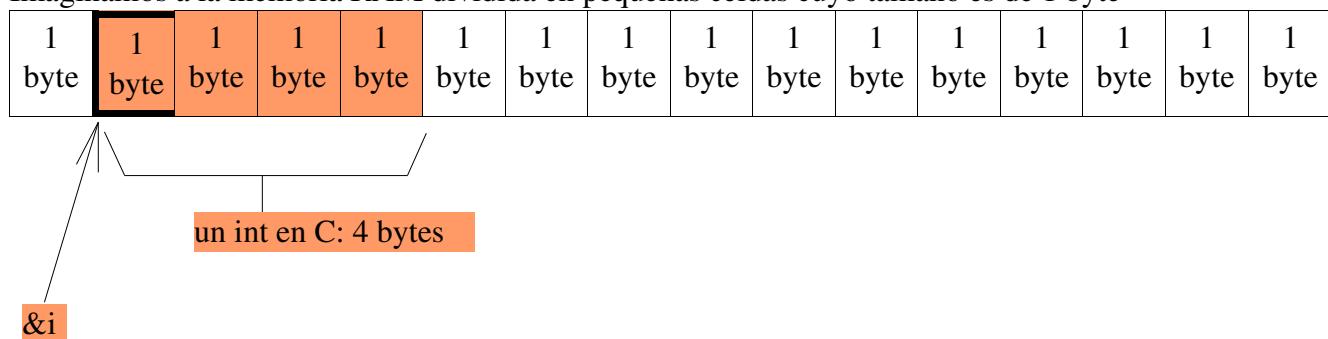
El Lenguaje de programación **C** permite averiguar cuál es la dirección de una variable mediante el **operador de dirección o referencia: &**

Ejemplo: veamos el siguiente trozo de código fuente

```
int i; // En C un entero ocupa 4 bytes en memoria  
  
&i; // Entrega la dirección de memoria, utilizando un sistema de representación de  
// números cuya base es 16, sistema de representación hexadecimal (tiene 16  
// símbolos que usa para representar cualquier número, los símbolos son:  
// 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E y F) en la que se encuentra ubicado el entero i  
// En este caso retorna la dirección en memoria del primero de los 4 bytes que  
// conforman al entero i
```

Veamos el esquema aclaratorio:

Imaginamos a la memoria RAM dividida en pequeñas celdas cuyo tamaño es de 1 byte



A continuación veremos un ejemplo que hace uso de los punteros.

```
// Archivo: operadorReferencia.c
#include <stdio.h>
int main() // comienza el código principal ejecutable
{
    int i, j, k; // declara 3 variables de tipo entero
    i = j = k = 13; // inicializa el valor de las 3 variables en 13
    printf( "i = %d , j = %d , k = %d \n", i , j ,k ); // muestra el valor de las 3 variables
    printf( "&i = %p , &j = %p , &k = %p \n" , &i , &j , &k ); // muestra la dirección en memoria
    // de cada variable
    return 0; // salida del programa exitosa, todo ha salido bien
}
// Fin del Archivo: operadorReferencia.c
```

Hay que notar la correspondencia, con respecto al orden y los tipos de datos, que se debe respetar al utilizar las **especificaciones de conversión** y los **caracteres ordinarios** en la función “**printf**”(salida con formato).

Cada **especificación de conversión** comienza con un **%** y termina con un **carácter de conversión**. Entre el **%** y el **carácter de conversión** pueden presentarse las siguientes **indicaciones opcionales**, en el orden en que se las describe a continuación:

- **signo menos**: indica el **ajuste hacia la izquierda del argumento convertido** (carácter ordinario)
- **número**: indica el **ancho mínimo del campo que contiene al argumento convertido**. Si es necesario se llenará con espacios en blanco cuando sobre espacio.
- **punto**: separa el **ancho mínimo de la precisión**.
- **número**: indica la **precisión**. Este valor tiene **dos interpretaciones posibles**:
 - **Si el argumento convertido es un carácter (tipo char)**: la precisión indica el **número máximo de caracteres que serán impresos**.
 - **Si el argumento convertido es un numero de tipo**:
 - **int (entero)** : la precisión indica el **número mínimo de dígitos** para ese entero
 - **double o float**: la precisión indica el **número de dígitos después del punto decimal**
- **h**: si el argumento convertido es entero y será impreso como un dato de tipo “short”
- **l**: una “ele”, si el argumento convertido es entero y será impreso como un dato de tipo “long”

Veamos algunas de las **especificaciones de conversión** que se pueden utilizar con “**printf**”:

- **%d** : recibe un **dato de tipo int (entero)** y lo muestra en **formato decimal**
- **%p**: recibe un **puntero** y lo muestra en **formato hexadecimal**
- **%x**: recibe un **dato de tipo int (entero)** y lo muestra en **formato hexadecimal**
- **%u**: recibe un **dato de tipo unsigned int (entero sin signo, positivo)** y lo muestra en **formato decimal**

Retomamos un trozo de código para aclarar estas cuestiones:

```
printf( " i = %d , j = %d , k = %d \n" , i , j , k );
```

El primer %d hace referencia al valor de i.

El segundo %d hace referencia al valor de j.

El tercer %d hace referencia al valor de k.

Se debe tener cuidado al utilizar las especificaciones de conversión en forma adecuada, porque pueden suceder errores de inconsistencia de tipos de datos, como por ejemplo en el siguiente código fuente:

```
// Archivo: errores-printf.c
#include <stdio.h>
int main(){
    int i = 10;
    double d = 2.5;
    printf( "\n d vale %d \n" , d );      // error de correspondencia de tipos

    printf( "\n d está en: %p \n" , d );
    // error de correspondencia de tipos, pues debe ir &d, no d

    return 0;    // salida satisfactoria del programa
}
// Fin del archivo: errores-printf.c
```

Este archivo se puede compilar con gcc de la siguiente manera:

```
gcc errores-printf.c -o errores-printf.out
```

El programa compilado se ejecuta de la siguiente manera (suponiendo que el archivo se encuentra en el mismo directorio en donde estamos ubicados actualmente):

```
./errores-printf.out
```

Cuya salida en pantalla es la siguiente:

```
d vale 0
```

```
d está en: (nil)
```

Obviamente se debe esto a que se están empleando mal las especificaciones de conversión, sin embargo el gcc nos permite compilar, esto se debe a que al trabajar con el Lenguaje de Programación C, el compilador realiza la menor cantidad de suposiciones posibles dejándole toda la responsabilidad en cuanto al control de la lógica embebida en el código al programador.

La versión lógicamente correcta es la siguiente:

```
// Archivo: errores-printf-corregido.c
#include <stdio.h>
int main(){
    int i = 10;
    double d = 2.5;
    printf( "\n d vale %f \n" , d );
    printf( "\n d está en: %p \n" , &d );

    return 0;    // salida satisfactoria del programa
}
// Fin del archivo: errores-printf-corregido.c
```

Este archivo **se puede compilar con gcc de la siguiente manera:**

gcc errores-printf-corregido.c -o errores-printf-corregido.out

El **programa compilado se ejecuta de la siguiente manera** (suponiendo que el archivo se encuentra en el mismo directorio en donde estamos ubicados actualmente):

./errores-printf-corregido.out

Cuya salida en pantalla es la siguiente:

d vale 2.500000

d está en: 0xbffff788

Ahora bien, si el tipo de dato de i es int. ¿Cuál es el tipo de dato del valor &i ?

El tipo de dato de &i es “puntero a entero” y esto se denota así: int *.

Por supuesto, sea:

double d; tiene tipo de dato double

&d tiene tipo de dato: “puntero a double” y se denota así: double *

En general, si tengo el tipo de dato T y sea v una variable del tipo de dato T, entonces:

T v; // v es una variable del tipo de dato T

&v; // tiene el tipo de dato “puntero a T” y se denota así: T *

Esta definición hace uso de la **notación hebrea: se lee mejor de derecha a izquierda**

Por lo tanto es evidente que C permite guardar punteros.

Por ejemplo:

```
// Archivo: guardamos-punteros.c
#include <stdio.h>
int main() // comienza la función principal ejecutable
{
    int i = 13;
    double d = 3.14;
    int * pi; // pi tiene el tipo de dato "puntero a entero"
    double * pd; // pd tiene el tipo de dato "puntero a double"
    pi = &i; // guarda la dirección en memoria de i en pi
    pd = &d; // guarda la dirección en memoria de d en pd
    printf("i vale: %d , y está en: %p \n" , i , pi );
    printf("d vale: %f , y está en: %p \n" , d , pd );
    return 0; // salida exitosa del programa
}
// Fin del archivo: guardamos-punteros.c
```

Peor aún, se pueden hacer cosas mas extrañas como en el siguiente código fuente en Lenguaje C:

```
// Archivo: guardamos-punteros-a-punteros.c
#include <stdio.h>
int main()
{
    int i , * pi;           // i es del tipo entero, pi es del tipo "puntero a entero"
    double d , * pd;        // d es del tipo double, pd es del tipo "puntero a double"
    int ** ppi; double ** ppd; // ppi es del tipo "puntero a puntero a entero"
                                // ppd es del tipo "puntero a puntero a double"
    i = 13 ; d = 3.14;
    pi = &i ; pd = &d;      // guarda la dirección en memoria de i en pi
                            // guarda la dirección en memoria de d en pd

    ppi = &pi ; ppd = &pd; // guarda la dirección en memoria de pi en ppi
                            // guarda la dirección en memoria de pd en ppd
    printf(" i vale %d , y está en %p \n" , i , pi );
    printf(" pi vale %p , y está en %p \n" , pi , ppi );
    printf(" ppi vale %p , y está en %p \n" , ppi , &ppi );
    printf("\n d vale %f , y está en %p \n" , d , pd );      // usa %f porque dato es double
    printf(" pd vale %p , y está en %p \n" , pd , ppd );
    printf(" ppd vale %p , y está en %p \n" , ppd , &ppd );
    return 0;      // salida exitosa del programa
}

// Fin del archivo: guardamos-punteros-a-punteros.c
```

Nuevamente este archivo se compila con:

gcc guardamos-punteros-a-punteros.c -o guardamos-punteros-a-punteros.out

y se ejecuta con:

./guardamos-punteros-a-punteros.out

cuya salida en pantalla es:

```
i vale 13 , y está en 0xbffff784
pi vale 0xbffff784 , y está en 0xbffff780
ppi vale 0xbffff780 , y está en 0xbffff770
```

```
d vale 3.140000 , y está en 0xbffff778
pd vale 0xbffff778 , y está en 0xbffff774
ppd vale 0xbffff774 , y está en 0xbffff76c
```

Pero al margen de saber manejar bien las direcciones de memoria, **lo que le interesa a un programador, es mas bien manipular el contenido o valor que está siendo referenciado mediante un puntero.** C permite este tipo de manipulación de datos mediante el **operador de indirección o de dereferenciación:** es el ***** y se lo utiliza antepuesto al puntero.

Por ejemplo si tenemos:

```
int i = 10, * pi = &i; // acá está declarando que pi es un puntero a entero que almacena la
                        // dirección en memoria de i
```

```
*pi; // retorna el valor que está almacenado en la dirección de memoria “pi”, es decir, en
      // la dirección de memoria de i, es decir 10
```

En general si tengo el **tipo de dato T** y sea **v una variable del tipo de dato T**, entonces:

T v; // v es una variable del tipo de dato T

&v; // tiene el tipo de dato “puntero a T” y se denota así: T *

T * pv = &v; // pv es una variable del tipo “puntero a T” que almacena la dirección de
// memoria de la variable v

*pv // toma al “puntero a T” pv y retorna el valor que está en la dirección de
// memoria indicada por pv, es decir, el valor de la variable v

Es de relevancia notar que:

- el puntero me dice “a partir de qué dirección”, y
- el tipo de dato del puntero me dice “cuánta memoria debe tomar”

Veamos esto con un ejemplo:

// Archivo: manipula-valor-mediante-punteros.c

```
#include <stdio.h>
int main()
{
    int i = 13, j = 12;
    int * pi = &i , * pj = &j ;
    printf("i = %d , j = %d \n" , i , j );
    (*pi)++; // al valor apuntado por pi sumarle uno
    (*pj)++; // al valor apuntado por pj sumarle uno
    printf("Ahora i = %d , y j = %d \n" , i , j);
    return 0;
}
```

// Fin del archivo: manipula-valor-mediante-punteros.c

Debe notarse la **inclusión de los paréntesis, en la sentencia: (*pi)++;**. Estos se han puesto para romper con las reglas de precedencia y asociatividad de los operadores que posee el Lenguaje C, las cuáles de detallan a continuación.

Reglas de precedencia y asociatividad de los operadores en el Lenguaje de Programación C:
 Desde la mayor precedencia (parte superior de la tabla) hacia la menor precedencia (parte inferior de la tabla)

OPERADORES	ASOCIATIVIDAD
() [] ->	de izquierda a derecha
! ~ ++ -- + - * & (tipo) sizeof	de derecha a izquierda
* / %	de izquierda a derecha
+ -	de izquierda a derecha
<< >>	de izquierda a derecha
< <= > >=	de izquierda a derecha
== !=	de izquierda a derecha
&	de izquierda a derecha
^	de izquierda a derecha
	de izquierda a derecha
&&	de izquierda a derecha
	de izquierda a derecha
?:	de derecha a izquierda
= += -= *= /= %= &= ^= = <<= >>=	de derecha a izquierda
,	de izquierda a derecha

Ahora veamos un ejemplo del “benemérito Lenguaje de Programación JAVA” que tiene relación con lo que estamos viendo acerca de punteros:

// Archivo: Q.java por convención termina con el sufijo “.java”

// En java el nombre del archivo, que posee el código fuente, debe ser el mismo que el que tiene
// la única clase pública que se permite definir dentro de su contenido. Notar que se está
// haciendo uso de una **técnica de diseño** denominada: **Diseño Orientado a Objetos** y una
// **técnica de Programación** denominada: **Programación Orientada a Objetos**

```
public class Q{
```

// Atributos de clase (static): no declarados

// Atributos de instancias de esta clase: no declarados

// Métodos de clase(static)

/* Método estático: swap función de intercambio

* Valor de retorno: no retorna ningún valor, se indica con void
* Parámetros de entrada: dos enteros
static void swap(int i , int j){

 int tmp = i;

 i = j;

 j = tmp;

}

// Métodos de instancias de esta clase: no declarados

// Constructor: no definido

// Método Principal de ejecución (main)

```
public static void main( String[] args){
```

 int i = 12, j = 13;

 // muestra en pantalla y hace nueva línea y retorno de carro con System.out.println

 System.out.println("i = " + i + " y j = " + j);

 swap(i , j);

 System.out.println("Ahora i = " + i + " y j = " + j);

}

}

// Fin del archivo: Q.java

Para poder compilar este archivo hay que tener instalado previamente el **“Java Development Kit: jdk”** que se puede obtener de la página web: <http://java.sun.com> . Se debe notar que este Lenguaje de Programación es un tanto más extraño que el Lenguaje C, no solo por su sintaxis, sino también por la manera en que se debe utilizar.

El jdk consta de tres componentes básicos:

- El compilador de JAVA, “JAVA COMPILER”: **javac**
- El intérprete de JAVA, “JAVA VIRTUAL MACHINE”: **java**
- Un conjunto de Librerías de clases para su utilización: “APPLICATION PROGRAMMING INTERFACE”, API, es decir la **INTERFACE DE PROGRAMACIÓN DE APLICACIONES**

Entonces es evidente que el Lenguaje de Programación **JAVA** es un **tipo de lenguaje “COMPILADO”** y también es un lenguaje **“INTERPRETADO”**.

Veremos a continuación cómo se compila y ejecuta el archivo Q.java :

/data/jdk1.5.0_04/bin/javac Q.java

Nota: Obviamente observar que usted tendrá que indicar el camino donde usted instaló el jdk en lugar de “/data/jdk1.5.0_04/bin/javac”

Esta instrucción para la línea de comandos lo que hace es realizar el **proceso de compilación del archivo de código fuente en Lenguaje JAVA**, pero el resultado de este proceso no es un archivo ejecutable por el microprocesador de la computadora, sino que se genera lo que se denomina un **“BYTECODE”**, es decir, un archivo que sólo es posible ejecutarlo mediante el intérprete de **JAVA**. El bytecode generado va a tener el nombre, en este caso, **Q.class** .

Por lo visto, para poder ejecutar este bytecode generado se debe utilizar:

/data/jdk1.5.0_04/bin/java Q

Nota: Nuevamente observar que usted tendrá que indicar el camino donde usted instaló el jdk en lugar de “/data/jdk1.5.0_04/bin/java” . **Notar que no es necesario indicar Q.class.**

La salida en pantalla es la siguiente:

i = 12 y j = 13

Ahora i = 12 y j = 13

Finalmente observando detenidamente bien la salida en pantalla **nos damos cuenta que la función swap, que debería intercambiar los valores, no esta realizando lo que nosotros queríamos.**

Veamos la versión de este código fuente pero ahora en Lenguaje C:

```
// Archivo: intercambio.c
#include <stdio.h>
// Prototipo de la función swap
int swap( int , int );

int main()
{
    int a = 12, b = 13;
    printf(" a = %d y b = %d \n", a , b );
    swap( a , b );
    printf("Ahora a = %d y b = %d \n", a , b );
}
// implementación de la función swap
int swap( int i , int j )
{
    int tmp = i;
    i = j;
    j = tmp;
}
// Fin del archivo: intercambio.c
```

Se compila con: **gcc intercambio.c -o intercambio.out**

Se ejecuta con: **./intercambio.out**

Su salida en pantalla es: **a = 12 y b = 13**
Ahora a = 12 y b = 13

Nuevamente observando la salida en pantalla **nos damos cuenta que la función swap, que debería intercambiar los valores, no está realizando lo que nosotros queremos.**

¿Por qué sucede esto? ¿Por qué sucede con C? y ¿Por qué sucede con JAVA?

Esto sucede porque tanto C, como JAVA realizan el PASAJE DE PARÁMETROS A LAS FUNCIONES POR VALOR, ES DECIR, UNA COPIA DE CADA VARIABLE SE PASA COMO PARÁMETRO A LA FUNCIÓN, Y NÓ LA REFERENCIA O DIRECCIÓN DE ESA VARIABLE.

ES POR ESTO QUE C POSEE PUNTEROS, PARA PERMITIR REALIZAR PASAJE DE PARÁMETROS A FUNCIONES POR REFERENCIA.

Así la versión lógicamente correcta de la función de intercambio se describe a continuación:

// Archivo: intercambio-corregido.c

```
#include <stdio.h>

/* Prototipo de la función swap. Tipo del valor de retorno: entero. Tipo de los Parámetros:
 * "puntero a entero"
 */
int swap( int * , int * );

int main() // Función principal ejecutable
{
    int a = 12, b = 13;
    int * pa = &a, * pb = &b;
    printf(" a = %d y b = %d \n", a , b );
    swap( pa , pb );
    printf("Ahora a = %d y b = %d \n", *pa , *pb );
}

// implementación de la función swap
int swap( int * pi , int * pj)
{
    int tmp = *pi;
    *pi = *pj;
    *pj = tmp;
}
// Fin del archivo: intercambio-corregido.c
```

Se compila con: **gcc intercambio-corregido.c -o intercambio-corregido.out**

Se ejecuta con: **./intercambio-corregido.out**

Su salida en pantalla es:
a = 12 y b = 13
Ahora a = 13 y b = 12

De esta manera se soluciona este inconveniente, pues SE PASA EL PARÁMETRO COMO UNA COPIA DE LA REFERENCIA O DIRECCIÓN EN MEMORIA DE ESAS VARIABLES.

Si queremos implementar una solución de este tipo **en el Lenguaje JAVA**, no podremos realizarla, pues **JAVA no permite la manipulación de datos mediante punteros, no se pueden definir punteros en JAVA.**

Ahora introducimos un nuevo concepto, la **conversión forzosa de tipos**, o en inglés, **CASTING**. C permite realizar conversión forzosa de tipos anteponiendo a una variable o expresión, el tipo al que se le quiere convertir entre paréntesis.

Al realizar CASTING se debe tener cuidado con los rangos y las precisiones de los valores que se están convirtiendo, por ejemplo:

```
// Archivo: casting.c
#include <stdio.h>
int main()
{
    double a = 10.56;
    printf("El double a = %f \n ", a);
    int b = (int) a;
    printf("Este mismo a convertido a entero vale: %d \n ", b );
    return 0;    // Salida exitosa
}
// Fin del archivo: casting.c
```

Se compila con: **gcc casting.c -o casting.out**

Se ejecuta con: **./casting.out**

Su salida en pantalla es:
El double a = 10.560000
Este mismo a convertido a entero vale: 10

Es evidente que se pierde la precisión, es decir los dígitos después del punto decimal, al convertir la variable a, que es del tipo double, al tipo int. Una vez más es responsabilidad del programador verificar estas cuestiones.

Veamos un poco de **Aritmética de punteros (aritmética de direcciones de memoria de variables de un tipo de dato dado)**:

La suma de un puntero y un entero da como resultado otro puntero al mismo tipo de dato

En general, sea:

T	un tipo de dato
T * p	p un “puntero a T”
int n	n un entero (de cualquier valor dentro del rango permitido)

entonces:

$$p + n \quad \text{es equivalente a} \quad (\text{unsigned int})p + n * \text{sizeof}(T)$$

NOTA: ES MUY IMPORTANTE NOTAR QUE EL CASTING EN LA RELACIÓN ANTERIOR SOLO AFECTA A p.

Para poder ver en toda su extensión este concepto, indagaremos en **otra característica del Lenguaje de Programación C: los ARREGLOS**.

Se trata de una **estructura de datos básica** que está soportada nativamente por el Lenguaje de Programación C. Su finalidad es el **almacenamiento contiguo de datos de un mismo tipo en memoria**.

Se accede a cada elemento del arreglo mediante un índice. Los **índices de los arreglos en C cumplen la siguiente relación:**

$$0 \leq \text{indice} \leq (\text{tamaño del arreglo} - 1)$$

Para declarar un arreglo se realiza de la siguiente manera:

TIPO-de-DATO nombre-del-arreglo[tamaño];

opcionalmente se puede definir e inicializar simultáneamente los elementos del arreglo de la siguiente manera:

TIPO-de-DATO nombre-del-arreglo[] = { valor_0 , valor_1 , ... , valor_tamaño-1 };

De otra manera, se debería, primero declararlo y luego inicializar cada valor por separado:

TIPO-de-DATO nombre-del-arreglo[tamaño];

nombre-del-arreglo[0] = valor_0;

nombre-del-arreglo[1] = valor_1;

.

.

.

nombre-del-arreglo[tamaño - 1] = valor_tamaño-1

Ahora veamos el siguiente archivo que contiene código fuente en Lenguaje C sobre estos temas que acabamos de ver:

```
// Archivo: aritmetica-punteros.c
```

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

```
int main()
```

```
{
```

```
    int arregloenteros[] = { 1 , 2 , 3 , 4};
```

```
    double arreglodoubles[] = { 11.11 , 22.22 , 33.33 , 44.44};
```

```
    int * parregloenteros = &arregloenteros[0];
```

```
    double * parreglodoubles = &arreglodoubles[0];
```

```
// Continúa el archivo: aritmetica-punteros.c
printf("El entero arregloenteros[0] = %d ", *parregloenteros );
printf("y se encuentra en la dirección de memoria: %p \n", parregloenteros );
printf("El espacio que ocupa en memoria el entero arregloenteros[0] es: %d bytes\n", sizeof( int ) );
printf("\nEl double arreglodoubles[0] = %f ", *parreglodoubles );
printf("y se encuentra en la dirección de memoria: %p \n", parreglodoubles );
printf("El espacio que ocupa en memoria el double arreglodoubles[0] es: %d bytes\n", sizeof(double));

/* En cada printf vamos a incrementar los punteros, sumándole uno en cada invocación siguiente y
 * usando el operador de indirección (*) para mostrar los valores que hay en esas direcciones de
 * memoria.
*/
parregloenteros++;

parreglodoubles++;

printf("\n arregloenteros[1]= %d ", *parregloenteros );
printf("y se encuentra en la dirección de memoria: %p \n", parregloenteros );
printf("\n arreglodoubles[1] = %f ", *parreglodoubles );
printf("y se encuentra en la dirección de memoria: %p \n", parreglodoubles );

parregloenteros++;

parreglodoubles++;

printf("\n arregloenteros[2]= %d ", *parregloenteros );
printf("y se encuentra en la dirección de memoria: %p \n", parregloenteros );
printf("\n arreglodoubles[2] = %f ", *parreglodoubles );
printf("y se encuentra en la dirección de memoria: %p \n", parreglodoubles );

parregloenteros++;

parreglodoubles++;

printf("\n arregloenteros[3]= %d ", *parregloenteros );
printf("y se encuentra en la dirección de memoria: %p \n", parregloenteros );
printf("\n arreglodoubles[3] = %f ", *parreglodoubles );
printf("y se encuentra en la dirección de memoria: %p \n", parreglodoubles );

/* Ahora veremos que las siguientes equivalencias entre direcciones de memoria es cierta:
 * parregloenteros + 1 <==> (unsigned)parregloenteros + 1 * sizeof( int )
 * parreglodoubles + 1 <==> (unsigned)parreglodoubles + 1 * sizeof( double )
*/
parregloenteros = &arregloenteros[0];

parreglodoubles = &arreglodoubles[0];

printf("arregloenteros[0] = %d ", *parregloenteros );
```

```
// Continúa el archivo: aritmetica-punteros.c
printf("en dirección de memoria: %p \n" , parregloenteros );
printf("arreglodoubles[0] = %f " , *parreglodoubles );
printf("en dirección de memoria: %p \n" , parreglodoubles );

parregloenteros = (unsigned)parregloenteros + 1 * sizeof( int ) ;
parreglodoubles = (unsigned)parreglodoubles + 1 * sizeof( double ) ;

printf("arregloenteros[1] = %d " , *parregloenteros );
printf("en dirección de memoria: %p \n" , parregloenteros );
printf("arreglodoubles[1] = %f " , *parreglodoubles );
printf("en dirección de memoria: %p \n" , parreglodoubles );

parregloenteros = (unsigned)parregloenteros + 1 * sizeof( int ) ;
parreglodoubles = (unsigned)parreglodoubles + 1 * sizeof( double ) ;

printf("arregloenteros[2] = %d " , *parregloenteros );
printf("en dirección de memoria: %p \n" , parregloenteros );
printf("arreglodoubles[2] = %f " , *parreglodoubles );
printf("en dirección de memoria: %p \n" , parreglodoubles );

parregloenteros = (unsigned)parregloenteros + 1 * sizeof( int ) ;
parreglodoubles = (unsigned)parreglodoubles + 1 * sizeof( double ) ;

printf("arregloenteros[3] = %d " , *parregloenteros );
printf("en dirección de memoria: %p \n" , parregloenteros );
printf("arreglodoubles[3] = %f " , *parreglodoubles );
printf("en dirección de memoria: %p \n" , parreglodoubles );

return 0; // salida exitosa del programa, todo ha salido bien
}

// Fin del archivo: aritmetica-punteros.c
```

Se compila con: **gcc aritmetica-punteros.c -o aritmetica-punteros.out**

Inmediatamente se observan los siguientes mensajes de advertencia:

aritmetica-punteros.c: En la función `main':
aritmetica-punteros.c:80: aviso: asignación se crea un puntero desde un entero sin una conversión
aritmetica-punteros.c:81: aviso: asignación se crea un puntero desde un entero sin una conversión
aritmetica-punteros.c:88: aviso: asignación se crea un puntero desde un entero sin una conversión
aritmetica-punteros.c:89: aviso: asignación se crea un puntero desde un entero sin una conversión
aritmetica-punteros.c:96: aviso: asignación se crea un puntero desde un entero sin una conversión
aritmetica-punteros.c:97: aviso: asignación se crea un puntero desde un entero sin una conversión

Se ejecuta con: **./aritmetica-punteros.out**

Su salida en pantalla es la siguiente:

El entero arregloenteros[0] = 1 y se encuentra en la dirección de memoria: 0xbfffff790
El espacio que ocupa en memoria el entero arregloenteros[0] es: 4 bytes

El double arreglodoubles[0] = 11.110000 y se encuentra en la dirección de memoria: 0xbfffff770
El espacio que ocupa en memoria el double arreglodoubles[0] es: 8 bytes

arregloenteros[1]= 2 y se encuentra en la dirección de memoria: 0xbfffff794

arreglodoubles[1] = 22.220000 y se encuentra en la dirección de memoria: 0xbfffff778

arregloenteros[2]= 3 y se encuentra en la dirección de memoria: 0xbfffff798

arreglodoubles[2] = 33.330000 y se encuentra en la dirección de memoria: 0xbfffff780

arregloenteros[3]= 4 y se encuentra en la dirección de memoria: 0xbfffff79c

arreglodoubles[3] = 44.440000 y se encuentra en la dirección de memoria: 0xbfffff788
arregloenteros[0] = 1 en dirección de memoria: 0xbfffff790

arreglodoubles[0] = 11.110000 en dirección de memoria: 0xbfffff770

arregloenteros[1] = 2 en dirección de memoria: 0xbfffff794

arreglodoubles[1] = 22.220000 en dirección de memoria: 0xbfffff778

arregloenteros[2] = 3 en dirección de memoria: 0xbfffff798

arreglodoubles[2] = 33.330000 en dirección de memoria: 0xbfffff780

arregloenteros[3] = 4 en dirección de memoria: 0xbfffff79c

arreglodoubles[3] = 44.440000 en dirección de memoria: 0xbfffff788

Arreglos y punteros en Lenguaje C:

Un arreglo en C es una **región de memoria contigua que almacena valores de un mismo tipo.**

Por ejemplo:

```
int a[10];      // un arreglo de 10 enteros
```

```
a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]
a[8]
a[9]
```

Podemos visualizarlo de la siguiente forma, imaginando a la memoria RAM como se ve a continuación:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1
B B B B	B B B B	B B B B	B B B B	B B B B	B B B B	B B B B	B B B B	B B B B	B B B B
Y Y Y Y	Y Y Y Y	Y Y Y Y	Y Y Y Y	Y Y Y Y	Y Y Y Y	Y Y Y Y	Y Y Y Y	Y Y Y Y	Y Y Y Y
T T T T	T T T T	T T T T	T T T T	T T T T	T T T T	T T T T	T T T T	T T T T	T T T T
E E E E	E E E E	E E E E	E E E E	E E E E	E E E E	E E E E	E E E E	E E E E	E E E E

a | 1 int |

Si hacemos:

```
int a[5], * pa = &a[0];
```

Entonces las siguientes expresiones son idénticas:

a[0]	es equivalente a	*(pa + 0)
a[1]	es equivalente a	*(pa + 1)
a[2]	es equivalente a	*(pa + 2)
a[3]	es equivalente a	*(pa + 3)
a[4]	es equivalente a	*(pa + 4)

Lo asombroso es que C permite también las siguientes equivalencias:

*(a + 0)	es equivalente a	pa[0]
*(a + 1)	es equivalente a	pa[1]
*(a + 2)	es equivalente a	pa[2]
*(a + 3)	es equivalente a	pa[3]
*(a + 4)	es equivalente a	pa[4]

Lo que nos lleva a inferir que: **¡UN ARREGLO ES UN PUNTERO!**

Veamoslo mejor con un ejemplo, pero antes necesitamos introducir **otra característica del Lenguaje C: El Bucle de repetición for (“PARA”)**

Esta es una sentencia que se utiliza para realizar bucles o iteraciones. Se denomina "for" (en español sería "PARA"). **Se la utiliza cuando se sabe de antemano la cantidad de veces que se va a repetir una cierta operación o sentencia.** Se usa mucho para la **manipulación de arreglos**. El bucle for tiene la siguiente sintaxis que se debe respetar:

TIPO-de-DATO iterador;

```
for( inicialización-iterador ; comparación-sobre-iterador ; incremento-decremento-del-iterador )
{
    sentencia-en-lenguaje-C-que-se-debe-ejecutar-en-cada-iteración;
    puede-ser-más-de-una-sentencia;
}
```

Es decir, la **inicialización del iterador se realiza una sola vez, antes de empezar el bucle de repetición de sentencias.**

Así para el iterador que parte desde un valor inicial, mientras el iterador siga cumpliendo la condición de la comparación, se ejecutan las sentencias, e inmediatamente después se incrementa, o decremente según corresponda, el valor del iterador. Si el iterador sigue cumpliendo con la condición de comparación, se ejecutan nuevamente las sentencias, sino se termina el bucle de repetición.

Dicho esto pasamos a ver el código fuente:

```
// Archivo: arreglos-son-punteros-y-viceversa.c
#include <stdio.h>
int main()
{
    printf("Declaramos un arreglo de 5 enteros: int a[5] .\n");
    int a[5];
    printf("Inicializamos los valores del arreglo a .\n");
    a[0] = 0;
    a[1] = 1;
    a[2] = 2;
    a[3] = 3;
    a[4] = 4;
    int indice = 0;
    for( indice = 0 ; indice < 5 ; indice++ )
    {
        printf("      a[%d] = %d      \n", indice , indice );
    }
    printf("Declaramos un \"puntero a arreglo de 5 enteros\": int * pa = &a[0] \n");
    int * pa = &a[0];
    printf("Veamos que se consigue lo mismo al mostrar a[indice] que al mostrar *(pa + indice) \n");
    for( indice = 0 ; indice < 5 ; indice++ )
    {
        printf("a[%d] = %d es igual que: *(pa + %d ) = %d \n" , indice , a[indice] , indice , *(pa + indice));
    }

    printf("Mas asombroso aún es lo siguiente: \n");

    for( indice = 0 ; indice < 5 ; indice++ )
    {
        printf("*(a + %d) = %d es igual que: pa[%d] = %d \n" , indice , *(a + indice), indice , pa[indice]);
    }

    printf("POR LO TANTO HEMOS VISTO QUE LOS ARREGLOS SON PUNTEROS\n");
    printf("Y HEMOS VISTO QUE LOS PUNTEROS SON ARREGLOS\n");
    return 0; // salida exitosa del programa, todo ha salido bien
}
// Fin del archivo: arreglos-son-punteros-y-viceversa.c
```

Se compila con: **gcc arreglos-son-punteros-y-viceversa.c -o arreglos-son-punteros-y-viceversa.out**

Se ejecuta con: **./arreglos-son-punteros-y-viceversa.out**

Su salida en pantalla es la siguiente:

Declaramos un arreglo de 5 enteros: int a[5] .

Inicializamos los valores del arreglo a .

a[0] = 0

a[1] = 1

a[2] = 2

a[3] = 3

a[4] = 4

Declaramos un "puntero a arreglo de 5 enteros": int * pa = &a[0]

Veamos que se consigue lo mismo al mostrar a[indice] que al mostrar *(pa + indice)

a[0] = 0 es igual que: *(pa + 0) = 0

a[1] = 1 es igual que: *(pa + 1) = 1

a[2] = 2 es igual que: *(pa + 2) = 2

a[3] = 3 es igual que: *(pa + 3) = 3

a[4] = 4 es igual que: *(pa + 4) = 4

Mas asombroso aún es lo siguiente:

*(a + 0) = 0 es igual que: pa[0] = 0

*(a + 1) = 1 es igual que: pa[1] = 1

*(a + 2) = 2 es igual que: pa[2] = 2

*(a + 3) = 3 es igual que: pa[3] = 3

*(a + 4) = 4 es igual que: pa[4] = 4

POR LO TANTO HEMOS VISTO QUE LOS ARREGLOS SON PUNTEROS!!!

Y HEMOS VISTO QUE LOS PUNTEROS SON ARREGLOS!!!

Algunas aclaraciones acerca del editor para la consola vi “Visual Editor”:

El editor vi opera en dos modos:

- **modo de inserción:** en el cuál lo que se escribe va al archivo
- **modo de órdenes:** en el cuál lo que se escribe lo interpreta como órdenes que debe ejecutar.
Este es el modo en que se encuentra el vi apenas se lo invoca.

Para pasar del modo de órdenes al modo de inserción se puede utilizar una de las siguientes opciones:

- **i :** Pulsando la tecla “i” se inserta a partir de donde se encuentra el cursor actualmente
- **a:** Pulsando la tecla “a” se inserta a partir del carácter consecutivo siguiente al que se encuentra el cursor actualmente
- **o :** Pulsando la tecla “o” se inserta a partir de la línea inferior siguiente a la que se encuentra el cursor actualmente
- **O :** Pulsando la tecla “shift + o” (o mayúscula) se inserta a partir de la línea superior siguiente a la que se encuentra el cursor actualmente

Para pasar del modo de inserción al modo de órdenes se pulsa la tecla “Esc”

Estando en modo de órdenes para ingresar un comando se pulsa la tecla “shift + .” (es decir la tecla o combinación de teclas que produce los dos puntos “:”) seguido del comando en cuestión.

Veamos **algunos ejemplos de comandos que se le pueden pasar al vi estando en modo de órdenes (Esc : orden)**:

- Esc : w **Guarda el archivo que se está editando al disco rígido**
- Esc : ! gcc % -o xxx.out **Compila con gcc el archivo que se está editando y guarda el ejecutable generado con el nombre “xxx.out” . Notar que obviamente primero debe guardarse el archivo al disco.**
- Esc : ! gcc -Wall % -o xxx.out **Idem anterior, pero esta vez se le pasa la opción “-Wall” al compilador gcc cuya función es “Warning all”, es decir, advierta todo lo discutible.**
- Esc : q **Sale del vi si todo ha sido guardado con anterioridad**
- Esc : q! **Sale del vi sin guardar**
- Esc : shell **Deja al vi ejecutándose en segundo plano “background” y abre una nueva sesión en el shell o línea de comandos de GNU/Linux, que generalmente se denomina “bash”. La primera versión de este shell fue escrita por Steve Bourne y es por esto que lleva su nombre: “GNU Bourne-Again SHell (el Shell de Bourne otra vez, de GNU)” Para volver al vi, simplemente debe cerrarse la sesión en el shell mediante el comando exit .**

Volvamos con C.

El ANSI C NO permite definir funciones dentro de funciones, pero el GCC lo permite como una extensión, aunque esto no es standar.

C tiene **PROTOTIPOS** de funciones que indican cómo se deben usar las funciones, cuántos y de qué tipo son los argumentos que toma y si tiene valor de retorno, a qué tipo de dato corresponde éste (si no tiene valor de retorno entonces se lo simboliza con “void”). Los prototipos de las funciones **deben terminar con un punto y coma (;).**

Esto evita meteduras de pata como, por ejemplo:

```
int i = exit( 0 );      // exit NO DEVUELVE VALORES
printf(15);            // ¡el primer argumento de printf debe ser un formato!
double d = sin( 0 );   // ¡sin debe tomar un double, no un int!
```

Ejemplos de prototipos de funciones:

```
void exit( int );
printf( char * fmt , ... );
double sin( double );
```

Normalmente los **prototipos de las funciones standar** están definidos en los **ARCHIVOS DE CABECERAS** o “**HEADERS**” que por convención tienen sufijo “.h”.

¿ Qué pasa con nuestras funciones?

Pués, exactamente lo mismo. Veamos esto con un ejemplo:

```
// Archivo: prototipos.c
#include <stdio.h>

// Prototipo de la función f
double f( double , char );

int main()
{
    double d = f( 3.14 , 'a' );
    printf(" f( 3.14 , 'a' ) = %f \n" , d );
    return 0 ;      // salida exitosa del programa
}

double f( double a , char c )
{
    double tmp = a + (double) c;
    return tmp;
}
// Fin del archivo: prototipos.c
```

Se compila con: **gcc prototipos.c -o prototipos.out**

Se ejecuta con: **./prototipos.out**

Su salida en pantalla es: **f(3.14 , 'a') = 100.140000**

Nota: esto es correcto pués el **CÓDIGO ASCII** del carácter a es:

<u>Octal</u>	<u>Decimal</u>	<u>Hexadecimal</u>	<u>Carácter</u>
141	97	61	a

y la función f realiza la suma: **3.140000 + 97.0000 = 100.140000**

Obviamente esta información acerca del **CÓDIGO ASCII** la pueden obtener de la siguiente página del manual en línea de GNU/Linux que se consulta con el comando: **man ascii**

Veamos ahora una versión levemente modificada del código fuente anterior:

```
// Archivo: prototipos-dos.c
#include <stdio.h>
double f( double a , char c ){
    double tmp = a + (double) c;
    return tmp;
}
int main(){
    double d = f( 3.14 , 'a' );
    printf(" f( 3.14 , 'a' ) = %f \n" , d );
    return 0 ;      // salida exitosa del programa
}
// Fin del archivo: prototipos-dos.c
```

Se observa que **se puede evitar la declaración del prototipo de la función si la definimos antes de que sea invocada en la función principal main**, de esta forma la **definición de la función es el prototipo de la función**.

El Lenguaje de Programación C permite el uso de **modificadores en las declaraciones de variables**. **El modificador va antepuesto a la declaración de la variable**. Entre ellos veremos los siguientes modificadores de variables:

- **const** : este modificador **indica que la variable** sobre la que se aplica **se va a comportar como un valor de sólo lectura**, es decir, **una vez definido no se puede cambiar**.
- **volatile** : este modificador **indica que el valor de la variable** sobre la que se aplica **puede cambiar incluso aún sin intervención del proceso que la definió**. Esto evita optimizaciones demasiado agresivas.

A continuación veremos algunos ejemplos con respecto a estos nuevos aspectos.

```
// Archivo: modificadores.c
#include <stdio.h>
int main(){
    const int i=10;
    printf("const int i=10;\n");
    int j = 5;
    printf("int j = 5;\n");
    i = j;                                // ERROR
    printf("i = j debería dar error\n");
    return 0;    // salida exitosa del programa
}
// Fin del Archivo: modificadores.c
```

Al intentar compilar este archivo con: **gcc ./modificadores.c -o modificadores.out**

Se obtiene el siguiente **mensaje de error del gcc**:

modificadores.c: En la función `main':
modificadores.c:8: error: asignación de read-only variable `i'

Indica el número de línea donde se detectó el error.

Esto se complica un poco con los punteros, como veremos a continuación en el siguiente código fuente en Lenguaje C.

```
1 // Archivo: modificadores-y-punteros.c
2 #include <stdio.h>
3 int main(){
4     int i = 10, j = 5;
5     int * pi = &i;      // CORRECTO
6     pi = &j;          // CORRECTO
7     *pi = 0;          // CORRECTO
8
9     const int * pj = &j;
10    // CORRECTO: El modificador const aplica a int primeramente,
11    // es decir, que lo que permanece constante es el valor del
12    // entero que está siendo apuntado.
13    // Se trata entonces de un puntero a una variable entera constante,
14    // un puntero a una variable entera de solo lectura.
15    // Observar que se puede cambiar el valor del puntero, pero no se
16    // puede cambiar el valor de la variable a la que apunta.
17
18    pj = &i;          // CORRECTO
19    // Se puede cambiar el valor del puntero (una dirección de memoria)
20
21    *pj = 20;         // ERROR
22    // No se puede cambiar el valor de la variable entera apuntada por ser const
23
24    int * const pk = &i;
25    // CORRECTO: El modificador const aplica a pk primeramente, es decir,
26    // que lo que permanece constante es el valor del puntero, por lo tanto
27    // se trata de un puntero que apunta siempre a la misma dirección de
28    // memoria, mientras que el valor de la variable entera que está siendo
29    // apuntada puede cambiar.
30
31    pk = &j;          // NO SE PUEDE CAMBIAR EL VALOR DEL PUNTERO
32    // (LA DIRECCIÓN DE MEMORIA A LA QUE APUNTA)
33    // PORQUE ES const
34
35    *pk = 30;         // CORRECTO
36
37    const int * const pl = &i;
38    // CORRECTO: Se trata del caso extremo. Un puntero con valor constante,
39    // que apunta siempre a la misma dirección de memoria, y cuya variable
40    // entera apuntada también es constante, es decir, la variable entera
41    // no se puede modificar.
42
43    pl = &j ;        //ERROR
44
45    *pl = 40 ;       //ERROR
46
47    return 0;
48 }
49 // Fin del archivo: modificadores-y-punteros.c
```

Se intenta compilar con: **gcc modificadores-y-punteros.c -o modificadores-y-punteros.out**

Pero obviamente el gcc muestra los siguientes mensajes de error:

modificadores-y-punteros.c: En la función `main':

modificadores-y-punteros.c:21: error: asignación of read-only location

modificadores-y-punteros.c:31: error: asignación of read-only variable `pk'

modificadores-y-punteros.c:43: error: asignación of read-only variable `pl'

modificadores-y-punteros.c:45: error: asignación of read-only location

Ahora que conocemos los modificadores de acceso, **podemos asegurar con un poco más de precisión que un arreglo es puntero constante**, cuyos valores, o sea las **direcciones de memoria, permanecen inalterables, pero los valores de las variables localizadas en esas direcciones si se pueden modificar.**

De este modo podemos ver que cuando el gcc reconoce la siguiente expresión:

int a[5]; // Un arreglo de 5 enteros

en realidad lo traduce a esta expresión equivalente: **int * const a**

que nosotros lo entendemos como: **“reservar espacio contiguo en memoria para 5 enteros”**

Veamos ahora para que puede servir el uso del modificador **volatile**.

Supongamos que estamos codificando un programa que controla un dispositivo, al cuál se lo activa cuando se envía el valor 1 (uno) en la dirección de memoria 100, y se lo desactiva enviando el valor 0 (cero) en la misma dirección de memoria.

```
// Archivo: volatile.c
#include <stdio.h>
int main()
{
    char * dispositivo = (char *) 100;
    // Aquí el casting se lo emplea para indicar que se trata de una
    // dirección de memoria. Suponemos que el dispositivo tiene un
    // ancho de palabra de 1 carácter = 1 byte = 8 bits

    *dispositivo = 1;      // indica arranque

    // Se supone que el valor apuntado por dispositivo va a cambiar
    // debido a un proceso externo al programa, y propio del
    // dispositivo

    while( *dispositivo != 0 )
    {
        printf("DISPOSITIVO FUNCIONANDO OK!\n");
    }

    return 0;      // Salida exitosa del programa
}
```

// Fin del archivo: volatile.c

Se compila con:

gcc volatile.c -o volatile.out

Se intenta ejecutar con:

./volatile.out

Pero se obtiene el siguiente mensaje de error:

Violación de segmento

Esto sucede porque se está queriendo hacer uso de la dirección de memoria 100 que por lo general está dentro del rango de las direcciones de memorias ocupadas por el Sistema Operativo. Se debe tener bien en claro que el sistema operativo está siempre ocupando una porción específica de memoria RAM, casi siempre en las primeras direcciones de memoria.

Una versión ejecutable del mismo ejemplo anterior podría ser el siguiente código fuente:

```
// Archivo: volatile.c
#include <stdio.h>
int main()
{
    char control = 1;
    char * pcontrol = &control;

    *pcontrol = 1; //indica arranque

    // Se supone que el valor apuntado por pcontrol va a cambiar
    // debido a un proceso externo al programa, y propio del
    // dispositivo

    while( (*pcontrol) != 0 )
    {
        printf("DISPOSITIVO FUNCIONANDO OK!\n");
        printf("PULSE LA TECLA 0 CERO PARA DESACTIVARLO!\n");
        printf("PULSE LA TECLA 1 UNO PARA QUE CONTINÚE FUNCIONADO!\n");
        scanf( "%d" , pcontrol );
        if( (*pcontrol) == 0 ) break;
    }

    return 0; // Salida exitosa del programa
}
// Fin del archivo: volatile.c
```

Se compila con:

gcc volatile.c -o volatile.out

Se ejecutar con:

./volatile.out

En cada repetición se muestra en pantalla:

```
DISPOSITIVO FUNCIONANDO OK!
PULSE LA TECLA 0 CERO PARA DESACTIVARLO!
PULSE LA TECLA 1 UNO PARA QUE CONTINÚE FUNCIONADO!
```

Si pulsa la tecla cero entonces termina la ejecución

Si pulsa la tecla uno entonces continúa la ejecución

Si pulsa **cualquier otra tecla** entonces nunca termina la ejecución, **entra en un bucle de ejecución infinito** del cuál sólo puede salir si presiona la combinación de teclas “**Ctrl + C**”.

Pero, ¿dónde aparece “volatile” en los códigos anteriores?

Ahora veremos para qué puede ser necesario usar el cualificador **volatile**. Como **gcc es un compilador optimizador**, resulta que al **compilar** el archivo anterior **gcc va a reemplazar el bucle while por la declaración de etiquetas que van a ser utilizadas por la sentencia “goto” de forma que resulte la misma iteración**. Así que para prevenir esta situación se debe hacer uso del **cualificador volatile**, como se muestra a continuación:

```
// Archivo: volatile-dos.c
#include <stdio.h>
int main()
{
    char control = 1;
    volatile char * pcontrol = &control;

    *pcontrol = 1; //indica arranque

    // Se supone que el valor apuntado por pcontrol va a cambiar
    // debido a un proceso externo al programa, y propio del
    // dispositivo

    while( (*pcontrol) != 0 )
    {
        printf("DISPOSITIVO FUNCIONANDO OK!\n");
        printf("PULSE LA TECLA 0 CERO PARA DESACTIVARLO!\n");
        printf("PULSE LA TECLA 1 UNO PARA QUE CONTINÚE FUNCIONADO!\n");
        scanf( "%d" , pcontrol );
        if( (*pcontrol) == 0 ) break;
    }

    return 0; // Salida exitosa del programa
}
// Fin del archivo: volatile-dos.c
```

Se compila con:

gcc volatile-dos.c -o volatile-dos.out

Se ejecutar con:

./volatile-dos.out

En cada repetición se muestra en pantalla:

```
DISPOSITIVO FUNCIONANDO OK!
PULSE LA TECLA 0 CERO PARA DESACTIVARLO!
PULSE LA TECLA 1 UNO PARA QUE CONTINÚE FUNCIONADO!
```

Si pulsa la tecla cero entonces termina la ejecución

Si pulsa la tecla uno entonces continúa la ejecución

Nuevamente si pulsa **cualquier otra tecla** entonces nunca termina la ejecución, **entra en un bucle de ejecución infinito** del cuál sólo puede salir si presiona la combinación de teclas “**Ctrl + C**”.

Una pregunta interesante es la siguiente:

¿En qué orden se almacenan los bytes dentro de un entero?

Supongamos que poseemos un microprocesador que maneja palabras de 32 bits.

Recordemos que un **entero, en Lenguaje C**, está **formado por 4 (CUATRO) bytes** (1 byte = 8 bits). Así que **de un entero se puede decir** que posee:

- **4 bytes**, o lo que es equivalente
- **8 * 4 = 32 dígitos binarios**, o lo que es equivalente
- **8 dígitos hexadecimales** (en base 16)

Nos interesa esta última equivalencia. **Por convención los dígitos hexadecimales tienen el prefijo: 0x**
Entonces un entero puede contener el siguiente valor en hexadecimal: 0x12345678

Pero, ¿Cómo se guarda 0x12345678?

Esto **dependerá de cuál sea el microprocesador** con el que estamos trabajando. Por lo general **podemos diferenciar a los microprocesadores de acuerdo a esta característica** como se menciona a continuación:

- **“BIG ENDIAN”**: Estos microprocesadores organizan el entero 0x12345678 como se grafica a continuación:

12	34	56	78
----	----	----	----

Son ejemplos de este tipo de microprocesadores: el MOTOROLA MC68X00, el VAX11, el SPARC, MIPS, PPC, etc.

- **“LITTLE ENDIAN”**: Estos microprocesadores organizan el entero 0x12345678 como se grafica a continuación:

78	56	34	12
----	----	----	----

Son ejemplos de este tipo de microprocesador: el INTEL X86, MIPS, PPC

- **“PDP/11”**: Este microprocesador de la empresa [Digital Equipment Corp.](#) organiza el entero 0x12345678 como se grafica a continuación:

56	78	12	34
----	----	----	----

Vamos a **hacer un programa para ver si nuestro sistema es “big endian” o “little endian”**.

```
// Archivo: big-o-little.c
#include <stdio.h>
int main()
{
    int i = 0x12345678;

    unsigned char * pi = (unsigned char *) &i;

    if( pi[0] == 0x12 && pi[1] == 0x34 && pi[2] == 0x56 && pi[3] == 0x78 )
    {
        printf("Este microprocesador es BIG ENDIAN ! \n");
    }
    else if( pi[0] == 0x78 && pi[1] == 0x56 && pi[2] == 0x34 && pi[3] == 0x12 )
    {
        printf("Este microprocesador es LITTLE ENDIAN ! \n");
    }
    else
    {
        printf("Este microprocesador es un PDP/11 ! \n");
    }
    return 0; // salida exitosa del programa
}
// Fin del archivo: big-o-little.c
```

Se compila con:

gcc big-o-little.c -o big-o-little.out

Se ejecuta con:

./big-o-little.out

Y la salida en pantalla para mi computadora fué: **Este microprocesador es LITTLE ENDIAN !**

Veamos ahora **una de las características más importantes del Lenguaje C**, que le da la connotación de un Lenguaje estructurado, la capacidad de poder definir y manipular **ESTRUCTURAS DE DATOS**.

¿Qué es una estructura de datos?

Una **estructura de datos** es una **colección de una o más variables**, de tipos posiblemente diferentes, **agrupadas bajo un solo nombre** para manejo conveniente. Las estructuras ayudan a organizar datos complicados, en particular dentro de programas grandes, debido a que permiten que a un grupo de variables relacionadas se las trate como una unidad en lugar de como entidades separadas. Una estructura es **un nuevo tipo de dato construido a partir de otros tipos de datos mas sencillos y ya definidos por el Lenguaje de Programación C**.

Para **declarar una estructura de datos** en C se hace uso de la **palabra reservada struct**, que puede estar **seguida de un nombre optativo llamado rótulo de estructura**, y a continuación, **entre llaves una lista de declaraciones de variables que componen a esta estructura, finalizando con un punto y coma: “;”**. El rótulo de estructura da nombre a esta clase de estructura, y en adelante puede ser utilizado como una abreviatura para la parte de declaraciones entre llaves.

Las **variables nombradas dentro de la estructura se llaman miembros**.

Una declaración struct define un tipo.

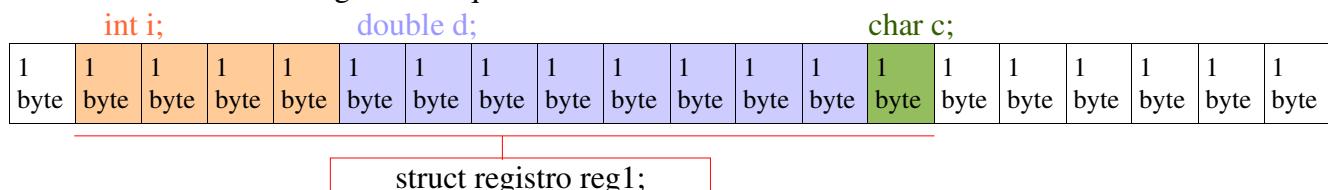
Una **declaración de estructura que no está seguida por una lista de variables no reserva espacio de almacenamiento** sino que simplemente **describe una plantilla o la forma de una estructura**. Sin embargo, si la declaración está rotulada, el rótulo se puede emplear posteriormente en definiciones de instancias de la estructura.

Una **estructura se puede inicializar al seguir su definición con una lista de inicializadores entre llaves, cada uno debe ser una expresión constante con el valor correspondiente a cada miembro de la estructura y separados por coma**.

Por ejemplo, veamos:

```
struct registro{           // Está declarando una estructura de tipo “struct registro”
    int i;                 // compuesta por: un entero, un double y un char.
    double d;
    char c;
} reg1 = { 1 , 1.1 , '1' }; // Declara la variable reg1 que es del tipo “struct registro”
                           // Inicializa los valores de las variables miembro
```

Lo visualizamos con el siguiente esquema:



Se hace referencia a un miembro de una estructura en particular en una expresión con una construcción de la forma:

nombre-estrucutra.miembro

El operador miembro de estructura “.” conecta el nombre de la estructura con el nombre del miembro.

Siguiendo con el ejemplo, tendríamos:

reg1.i	// para acceder al entero i de la estructura reg1
reg1.d	// para acceder al double d de la estructura reg1
reg1.c	// para acceder al char c de la estructura reg1

Las estructuras pueden anidarse, es decir, al definir estructuras se pueden usar estructuras definidas previamente como variables miembro.

Obviamente se puede obtener la dirección de memoria en la que se ubica la estructura con & y almacenarlas en “punteros a estructura”, así como las direcciones de las variables miembros.

Veamos el siguiente ejemplo:

```
// Archivo: estructuras.c
#include <stdio.h>

struct registro {
    int i; double d; char c;
};

int main()
{
    // Se declaran e inicializan variables del tipo "struct registro"
    struct registro reg1 = { 1 , 1.1 , '1' } , reg2 = { 2 , 2.2 , '2' };

    /* Se declaran e inicializan variables del tipo:
     * puntero a "struct registro" */
    struct registro * preg1 = &reg1 , * preg2 = &reg2 ;

    /* Se declaran e inicializan variables del tipo: "puntero a entero",
     * "puntero a double" , "puntero a char" para almacenar las
     * direcciones de las variables miembros de las estructuras */
    int * preg1_i = &( reg1.i ) , * preg2_i = &( reg2.i );
    double * preg1_d = &( reg1.d ) , * preg2_d = &( reg2.d );
    char * preg1_c = &( reg1.c ) , * preg2_c = &( reg2.c );

    printf("\nLa estructura de tipo \"struct registro\": reg1 está en: %p\n" , preg1);
    printf("Y está conformada por: \n");
    printf("\nun entero i: reg1.i = %d " , reg1.i );
    printf("que está en: &(reg1.i) = %p \n" , preg1_i );
    printf("el mismo entero i: (*preg1).i = %d " , (*preg1).i );
    printf("que está en: &((*preg1).i) = %p \n" , &((*preg1).i) );
    printf("el mismo entero i: preg1->i = %d " , preg1->i );
    printf("que está en: &(preg1->i) = %p \n" , &(preg1->i) );
    printf("\nun double d: reg1.d = %f " , reg1.d );
    printf("que está en: &(reg1.d) = %p \n" , preg1_d );
    printf("el mismo double d: (*preg1).d = %f " , (*preg1).d );
    printf("que está en: &((*preg1).d) = %p \n" , &((*preg1).d) );
```

//continúa el archivo: estructuras.c

```
printf("el mismo double d: preg1->d = %f ", preg1->d );
printf("que está en: &(preg1->d) = %p \n" , &(preg1->d ) );
printf("\nun char c: reg1.c = %c ", reg1.c );
printf("que está en: &(reg1.c) = %p \n" , preg1_c );
printf("el mismo char c:(*preg1).c = %c ", (*preg1).c );
printf("que está en: &((*preg1).c) = %p \n" , &((*preg1).c) );
printf("el mismo char c: preg1->c = %c ", preg1->c );
printf("que está en: &(preg1->c) = %p \n" , &(preg1->c) );
printf("\nLa estructura de tipo \"struct registro\": reg2 está en: %p\n" , preg2);
printf("Y está conformada por: \n");
printf("\nun entero i: reg2.i = %d ", reg2.i );
printf("que está en: &(reg2.i) = %p \n" , preg2_i );
printf("el mismo entero i: (*preg2).i = %d ", (*preg2).i );
printf("que está en: &((*preg2).i) = %p \n" , &((*preg2).i) );
printf("el mismo entero i: preg2->i = %d ", preg2->i );
printf("que está en: &(preg2->i) = %p \n" , &(preg2->i) );
printf("\nun double d: reg2.d = %f ", reg2.d );
printf("que está en: &(reg2.d) = %p \n" , preg2_d );
printf("el mismo double d: (*preg2).d = %f ", (*preg2).d );
printf("que está en: &((*preg2).d) = %p \n" , &((*preg2).d) );
printf("el mismo double d: preg2->d = %f ", preg2->d );
printf("que está en: &(preg2->d) = %p \n" , &(preg2->d) );
printf("\nun char c: reg2.c = %c ", reg2.c );
printf("que está en: &(reg2.c) = %p \n" , preg2_c );
printf("el mismo char c: (*preg2).c = %c ", (*preg2).c );
printf("que está en: &((*preg2).c) = %p \n" , &((*preg2).c) );
printf("el mismo char c: preg2->c = %c ", preg2->c );
printf("que está en: &(preg2->c) = %p \n" , &(preg2->c) );

reg1.i = 3;
printf("Ahora modificamos:\n");
printf("    reg1.i = 3;    \n");
reg1.d = 3.3;
printf("    reg1.d = 3.3;  \n");
reg1.c = '3';
printf("    reg1.c = '3'; \n");

printf("Obteniendo:\n");
printf("reg1.i = %d <=> (*preg1).i = %d <=> preg1->i = %d \n", reg1.i , (*preg1).i , preg1->i );
printf("reg1.d = %f <=> (*preg1).d = %f <=> preg1->d = %f \n", reg1.d , (*preg1).d , preg1->d );
printf("reg1.c = %c <=> (*preg1).c = %c <=> preg1->c = %c \n", reg1.c , (*preg1).c , preg1->c );

(*preg1).i = 5;
printf("Ahora modificamos:\n");
printf("    (*preg1).i = 5; \n");
(*preg1).d = 5.5;
printf("    (*preg1).d = 5.5;   \n");
(*preg1).c = '5';
printf("    (*preg1).c = '5'; \n");
```

```
//continúa el archivo: estructuras.c
printf("Obteniendo:\n");
printf("reg1.i = %d <=> (*preg1).i = %d <=> preg1->i = %d \n", reg1.i , (*preg1).i , preg1->i );
printf("reg1.d = %f <=> (*preg1).d = %f <=> preg1->d = %f \n", reg1.d , (*preg1).d , preg1->d );
printf("reg1.c = %c <=> (*preg1).c = %c <=> preg1->c = %c \n", reg1.c , (*preg1).c , preg1->c );

preg1->i = 7;
printf("Ahora modificamos:\n");
printf("    preg1->i = 7; \n");
preg1->d = 7.7;
printf("    preg1->d = 7.7; \n");
preg1->c = '7';
printf("    preg1->c = '7'; \n");

printf("Obteniendo:\n");
printf("reg1.i = %d <=> (*preg1).i = %d <=> preg1->i = %d \n", reg1.i , (*preg1).i , preg1->i );
printf("reg1.d = %f <=> (*preg1).d = %f <=> preg1->d = %f \n", reg1.d , (*preg1).d , preg1->d );
printf("reg1.c = %c <=> (*preg1).c = %c <=> preg1->c = %c \n", reg1.c , (*preg1).c , preg1->c );

preg2->i = 4;
printf("Por último modificamos:\n");
printf("    preg2->i = 4; \n");
preg2->d = 4.4;
printf("    preg2->d = 4.4; \n");
preg2->c = '4';
printf("    preg2->c = '4'; \n");

printf("Obteniendo:\n");
printf("reg2.i = %d <=> (*preg2).i = %d <=> preg2->i = %d \n", reg2.i , (*preg2).i , preg2->i );
printf("reg2.d = %f <=> (*preg2).d = %f <=> preg2->d = %f \n", reg2.d , (*preg2).d , preg2->d );
printf("reg2.c = %c <=> (*preg2).c = %c <=> preg2->c = %c \n", reg2.c , (*preg2).c , preg2->c );

printf("\nPOR LO TANTO:\n");
printf("LA OPERACIÓN CON EL OPERADOR SELECTOR DE MIEMBRO ");
printf("PROPIO DE LAS ESTRUCTURAS:\n");
printf("\n    estructura.variableMiembro\n");
printf("\nES EQUIVALENTE A USAR:\n");
printf("\nLA OPERACIÓN CON EL OPERADOR \"/-\>\" ");
printf("PROPIO DE LOS PUNTEROS A ESTRUCTURAS:\n");
printf("\n    punteroAestructura -> variableMiembro\n");
printf("\nQUE TAMBIÉN ES EQUIVALENTE A USAR:\n");
printf("\n    (*punteroAestructura ).variableMiembro\n");
printf("\nYA SEA PARA MOSTRAR VALORES DE LAS VARIABLES MIEMBROS \n");
printf("ASÍ COMO PARA MODIFICARLAS.\n");
return 0; // salida exitosa del programa
}

// Fin del archivo: estructuras.c
```

Se compila: **gcc estructuras.c -o estructuras.out**
Se ejecuta con: **./estructuras.out**

Su salida en pantalla es la siguiente:

La estructura de tipo "struct registro": reg1 está en: 0xbffff7a0

Y está conformada por:

un entero i: reg1.i = 1 que está en: &(reg1.i) = 0xbffff7a0

el mismo entero i: (*preg1).i = 1 que está en: &((*preg1).i) = 0xbffff7a0

el mismo entero i: preg1->i = 1 que está en: &(preg1->i) = 0xbffff7a0

un double d: reg1.d = 1.100000 que está en: &(reg1.d) = 0xbffff7a4

el mismo double d: (*preg1).d = 1.100000 que está en: &((*preg1).d) = 0xbffff7a4

el mismo double d: preg1->d = 1.100000 que está en: &(preg1->d) = 0xbffff7a4

un char c: reg1.c = 1 que está en: &(reg1.c) = 0xbffff7ac

el mismo char c: (*preg1).c = 1 que está en: &((*preg1).c) = 0xbffff7ac

el mismo char c: preg1->c = 1 que está en: &(preg1->c) = 0xbffff7ac

La estructura de tipo "struct registro": reg2 está en: 0xbffff790

Y está conformada por:

un entero i: reg2.i = 2 que está en: &(reg2.i) = 0xbffff790

el mismo entero i: (*preg2).i = 2 que está en: &((*preg2).i) = 0xbffff790

el mismo entero i: preg2->i = 2 que está en: &(preg2->i) = 0xbffff790

un double d: reg2.d = 2.200000 que está en: &(reg2.d) = 0xbffff794

el mismo double d: (*preg2).d = 2.200000 que está en: &((*preg2).d) = 0xbffff794

el mismo double d: preg2->d = 2.200000 que está en: &(preg2->d) = 0xbffff794

un char c: reg2.c = 2 que está en: &(reg2.c) = 0xbffff79c

el mismo char c: (*preg2).c = 2 que está en: &((*preg2).c) = 0xbffff79c

el mismo char c: preg2->c = 2 que está en: &(preg2->c) = 0xbffff79c

Ahora modificamos:

reg1.i = 3;

reg1.d = 3.3;

reg1.c = '3';

Obteniendo:

reg1.i = 3 \Leftrightarrow (*preg1).i = 3 \Leftrightarrow preg1->i = 3

reg1.d = 3.300000 \Leftrightarrow (*preg1).d = 3.300000 \Leftrightarrow preg1->d = 3.300000

reg1.c = 3 \Leftrightarrow (*preg1).c = 3 \Leftrightarrow preg1->c = 3

Ahora modificamos:

(*preg1).i = 5;

(*preg1).d = 5.5;

(*preg1).c = '5';

Obteniendo:

reg1.i = 5 \Leftrightarrow (*preg1).i = 5 \Leftrightarrow preg1->i = 5

reg1.d = 5.500000 \Leftrightarrow (*preg1).d = 5.500000 \Leftrightarrow preg1->d = 5.500000

reg1.c = 5 \Leftrightarrow (*preg1).c = 5 \Leftrightarrow preg1->c = 5

Ahora modificamos:

preg1->i = 7;

preg1->d = 7.7;

preg1->c = '7';

Obteniendo:

reg1.i = 7 <=> (*preg1).i = 7 <=> preg1->i = 7

reg1.d = 7.700000 <=> (*preg1).d = 7.700000 <=> preg1->d = 7.700000

reg1.c = 7 <=> (*preg1).c = 7 <=> preg1->c = 7

Por último modificamos:

preg2->i = 4;

preg2->d = 4.4;

preg2->c = '4';

Obteniendo:

reg2.i = 4 <=> (*preg2).i = 4 <=> preg2->i = 4

reg2.d = 4.400000 <=> (*preg2).d = 4.400000 <=> preg2->d = 4.400000

reg2.c = 4 <=> (*preg2).c = 4 <=> preg2->c = 4

POR LO TANTO:

LA OPERACIÓN CON EL OPERADOR SELECTOR DE MIEMBRO PROPIO DE LAS ESTRUCTURAS:

estructura.variableMiembro

ES EQUIVALENTE A USAR:

LA OPERACIÓN CON EL OPERADOR "->" PROPIO DE LOS PUNTEROS A ESTRUCTURAS:

punteroAestructura -> variableMiembro

QUE TAMBIÉN ES EQUIVALENTE A USAR:

(*punteroAestructura).variableMiembro

YA SEA PARA MOSTRAR VALORES DE LAS VARIABLES MIEMBROS ASÍ COMO PARA MODIFICARLAS.

Observando esta salida en pantalla detenidamente, destamos:

- al declarar la estructura e inicializar sus variables miembro se está reservando espacio de almacenamiento contiguo en memoria
- Resultan equivalentes las siguientes formas de manipular los valores de las variables miembros de una estructura:
 - aplicando el operador miembro a la estructura (como por ejemplo en: **reg1.i = 3;**)
 - aplicando el operador “->” al “puntero a estructura” (como en: **preg1 -> i = 3;**)
 - aplicando el operador miembro a un “puntero a estructura que se le ha aplicado previamente el operador de indirección” (como por ejemplo en: **(*preg1).i = 3;**)

Estructuras y funciones:

Las únicas operaciones legales sobre una estructura son **copiarla o asignarla como unidad, tomar su dirección de memoria con &, y tener acceso a sus miembros.**

La copia y la asignación incluyen pasarlas como parámetros a funciones y también regresar valores de funciones.

Las **estructuras no se pueden comparar**. Una estructura se puede inicializar con una lista de valores constantes de miembros; una estructura automática también se puede inicializar con una asignación.

Veamos un ejemplo al respecto:

```
// Archivo: estructuras-y-funciones.c
#include <stdio.h>

struct registro {
    int i; double d; char c;
};

// prototipo de la función crearRegistro
struct registro crearRegistro( int , double , char );

// prototipo de la función mostrarRegistro
void mostrarRegistro( struct registro * );

/* Función: crearRegistro
 * Destinada a la creación de variables del tipo "struct registro"
 * Retorna : una variable del tipo "struct registro"
 * Parámetros que recibe: un entero, un double, un char
 */
struct registro crearRegistro( int i , double d , char c )
{
    struct registro tmp = { i , d , c };
    return tmp;
}

/* Función: mostrarRegistro
 * Destinada a imprimir en pantalla el registro
 * Retorna: void (nada)
 * Parámetros que recibe: un "puntero a struct registro"
 */
void mostrarRegistro( struct registro * preg )
{
    printf("\n-----\n");
    printf("Registro conformado por: \n");
    printf("    preg -> i = %d \n" , preg -> i );
    printf("    preg -> d = %f \n" , preg -> d );
    printf("    preg -> c = %c \n" , preg -> c );
    printf("\n-----\n");
}
```

```
// continúa el archivo: estructuras-y-funciones.c
int main()
{
    // Se declaran e inicializan variables del tipo "struct registro"
    struct registro reg1 = crearRegistro( 1 , 1.1 , '1' );
    struct registro reg2 = crearRegistro( 2 , 2.2 , '2' );

    /* Se declaran e inicializan variables del tipo:
     * puntero a "struct registro" */
    struct registro * preg1 = &reg1 , * preg2 = &reg2 ;

    mostrarRegistro( preg1 );
    mostrarRegistro( preg2 );

    return 0;      // salida exitosa del programa
}
// Fin del archivo: estructuras-y-funciones.c
```

Se compila con: **gcc estructuras-y-funciones.c -o estructuras-y-funciones.out**
Se ejecuta con: **./estructuras-y-funciones.out**
Su salida en pantalla es:

Registro conformado por:

preg -> i = 1
preg -> d = 1.100000
preg -> c = 1

Registro conformado por:

preg -> i = 2
preg -> d = 2.200000
preg -> c = 2

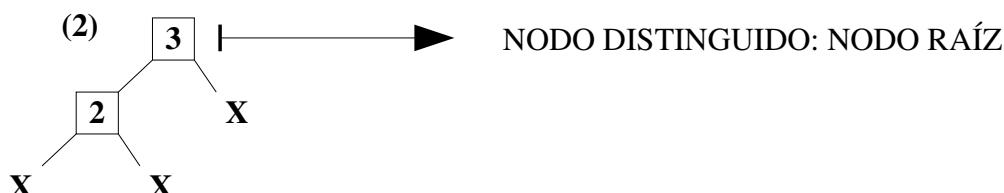
Ahora que conocemos las estructuras, veamos una que resulta interesante:

LOS ÁRBOLES BINARIOS.

Empecemos con un primer acercamiento sencillo.

Un **arbol binario** puede estar **formado sólo por su nodo distinguido denominado raíz**, en cuyo caso se lo denomina ARBOL NULO o ARBOL VACÍO:

(1) X ARBOL VACÍO: solo contiene un nodo raíz cuyo contenido es NULL



Un **arbol binario**, por lo general, tiene un **nodo raíz**, a partir del cuál se desprenden un **sub-arbol binario hacia la izquierda** y otro **sub-arbol binario hacia la derecha**. Por lo tanto un arbol binario es una estructura de datos que se define en forma recursiva, es decir **haciéndo referencia a sí misma durante su definición**.

Nos valdremos del **Lenguaje de programación ML** para realizar la **definición de un arbol binario**.

ML, acrónimo de **META LENGUAJE**, es un **Lenguaje de programación de propósito general** de la familia de los **lenguajes de programación funcional** desarrollado por **Robin Milner y otros** a **finales de los años 1970** en la **Universidad de Edimburgo**. Entre las características de ML se incluyen evaluación por valor, álgebra de funciones, manejo automatizado de memoria por medio de recolección de basura, polimorfismo parametrizado, análisis de estático de tipos, inferencia de tipos, tipos de datos algebraicos, llamada por patrones y manejo de excepciones.

Usemos ML para hacer un prototipo inicial de lo que será un arbol binario.

**datatype Arbol = Vacío |
Nodo of int * Arbol * Arbol**

Nota: | significa OR

Entonces para el ejemplo:

[3]
/ \
[2] X
/ \
X X

Nota: Las X representan que no hay elementos, solo una rama vacía.
La expresión para este arbol binario usando ML es:
Nodo(3 , Nodo(2 , Vacío , Vacío) , Vacío)

Con los árboles binarios se pueden hacer muchas operaciones.

Vamos a hacer una función para insertar “ordenadamente” un elemento (un número) en un arbol binario usando ML:

**fun inserta(n , Vacío) = Nodo(n , Vacío , Vacío) |
inserta(n , Nodo(n' , izq , der)) =
if n > n' then Nodo(n' , izq , inserta(n , der))
else if n < n' then Nodo(n' , inserta(n , izq) , der)
else Nodo(n , izq , der)**

Entonces al usar esta función, nos aseguramos que **al insertar un elemento al arbol binario, se respete el orden inherente a esta estructura y que no se presenten dos elementos repetidos en el arbol**. Así, sea un arbol cualquiera, si nos ubicamos en su raíz, podemos ver que **todos los elementos del sub-arbol izquierdo tienen valores inferiores al de la raíz**, mientras que **todos los elementos del sub-arbol derecho tienen valores superiores al de la raíz**.

Nuevamente usamos ML para una primera aproximación de lo que será la **función que recorre un arbol binario mostrando los valores que contienen sus nodos en forma ordenada**:

**fun recorre(Vacío) = () (* nada *) |
recorre(Nodo(n , izq , der)) =
(recorre izq;
print(makestring n);
recorre der;)**

Vamos a tratar de pasar esto a Lenguaje C.

El primer inconveniente que surge, por ser evidente, es que **C no tiene tipos recursivos como los que ML permite definir.**

Pues veamos que al querer compilar el siguiente código fuente en lenguaje C:

```
// Archivo: arbol-binario.c
#include <stdio.h>
struct Nodo{
    int n;
    struct Nodo izq, der;
}
int main()
{
    struct nodo raiz;
    return 0;
}
```

// Fin del archivo: arbol-binario.c

Con la orden: **gcc -Wall arbol-binario.c -o arbol-binario.out**

Se obtienen los siguientes mensajes de advertencias y errores, emitidos por el gcc:

```
arbol-binario.c:6: error: field `izq' has incomplete type
arbol-binario.c:6: error: field `der' has incomplete type
arbol-binario.c:10: error: two or more data types in declaration of `main'
arbol-binario.c: En la función `main':
arbol-binario.c:11: error: storage size of `raiz' isn't known
arbol-binario.c:11: aviso: unused variable `raiz'
```

Para vencer este inconveniente **vamos a usar “punteros a struct Nodo” para simular tipos recursivos en C.**

Denotamos “Vacío” con el “puntero nulo NULL”.

Todo anda mejor.

Ahora un **arbol binario** será un **“puntero a struct Nodo”**.

Pero como estamos diciendo que las **variables miembros** de nuestra estructura “**Nodo**” serán **“punteros a struct Nodo”** deberemos **hacernos cargo de reservar el suficiente espacio en memoria para almacenar esta estructura en forma correcta**, nosotros mismos **haciendo uso del operador “sizeof”, junto con la función “malloc”** que está **definida en el archivo “stdlib.h”, una función de la LIBRERÍA STANDAR DE C.**

El operador sizeof no es una función, es una palabra reservada, o sea, un operador que también tiene precedencia.

Ahora sí veamos el siguiente código fuente en Lenguaje C.

```
// Archivo: arbol-binario.c
#include <stdio.h>
#include <stdlib.h> // Para la función malloc. Funciones
// permiten asignar y liberar memoria dinámica

/* Declaración de la estructura Nodo para el arbol binario
 * Variables miembro:
 *   * entero           elemento del nodo
 *   * puntero a "struct Nodo"    sub-arbol izquierdo
 *   * puntero a "struct Nodo"    sub-arbol derecho
 */
struct Nodo{
    int elemento;
    struct Nodo * izq, * der;
};

// prototipo de la función inserta
struct Nodo * inserta( int , struct Nodo * );

// prototipo de la función recorre
void recorre( struct Nodo * );

/* Función: recorre
 * Destinada a mostrar en pantalla los elementos del arbol binario
 * en forma ordenada.
 * Retorna: void (nada)
 * Argumentos: puntero a "struct Nodo" a partir del cuál comenzará
 *               a mostrar sus elementos
 */
void recorre( struct Nodo * nodoElegido )
{
    if( nodoElegido != NULL )
    {
        recorre( nodoElegido->izq );
        printf( "\n %d \n" , nodoElegido -> elemento );
        recorre( nodoElegido->der );
    }
}

/* Función: inserta
 * Destinada a insertar un elemento al arbol a partir de un nodo
 * indicado mediante un "puntero a "struct Nodo"""
 * Retorna: un puntero a "struct Nodo"
 * Argumentos: un entero,    elemento que quiere insertar
 *   *      un puntero a "struct Nodo" a partir del cuál
 *   *      buscará el lugar adecuado para que al insertar
 *   *      el elemento, se mantenga el orden y no haya
 *   *      repetición de elementos.
 */
```

```
// Continúa el archivo: arbol-binario.c

struct Nodo * inserta( int elemento , struct Nodo * nodoElegido )
{
    if( nodoElegido == NULL ){

        nodoElegido = malloc( sizeof( struct Nodo ) );

        /* malloc asigna sizeof( struct Nodo ) bytes y
         * devuelve un puntero a la memoria asignada.
         * La memoria no es borrada.
         */
        nodoElegido -> elemento = elemento;
        nodoElegido -> izq = nodoElegido -> der = NULL;
    }
    else if( elemento < nodoElegido -> elemento ){
        nodoElegido -> izq = inserta( elemento , nodoElegido -> izq );
        // inserta el elemento en el sub-arbol izquierdo
    }
    else if( elemento > nodoElegido -> elemento ){
        nodoElegido -> der = inserta( elemento , nodoElegido -> der );
        // inserta el elemento en el sub-arbol derecho
    }
    return nodoElegido;
}

// Comienza el código ejecutable, función principal main
int main()
{
    struct Nodo * arbol = NULL;
    int indice;
    for( indice = 99 ; indice > 0 ; indice -= 2 )
    {
        arbol = inserta( indice , arbol );
    }
    recorre( arbol );
    return 0;    // salida exitosa del programa
}
// Fin del archivo: arbol-binario.c
```

Se compila con:

gcc arbol-binario.c -o ./arbol-binario.out

Se ejecuta con:

./arbol-binario.out

Su salida en pantalla es el conjunto de números impares [1 , 99] , incluídos los extremos, un número por línea. Junto con cada número se imprime una nueva línea en blanco antes y una nueva línea en blanco después de cada número.

Pero podemos agregar más funcionalidad al código anterior definiendo funciones que se encarguen de manipular los árboles binarios.

Por lo general lo que necesitamos hacer con un arbol binario es:

- insertar un elemento en el arbol binario
- buscar un elemento en el arbol binario
- borrar un elemento del arbol binario. Aquí se debe proceder con cuidado para conservar el orden de los elementos del árbol
- recorrer el arbol binario para mostrarlo en pantalla en forma ordenada
- encontrar el mayor elemento de arbol binario
- encontrar el menor elemento del arbol binario

Nos falta definir las funciones “busca”, “menor”, “mayor” y “borra”.

Nuevamente vamos a valernos de ML para realizar las primeras versiones de estas funciones:

La implementación de la función “busca” en Lenguaje ML es la siguiente:

```
fun busca( n , Vacío )          =      Vacío |  
busca( n , Nodo( n' , izq , der ) ) =  
    if n = n' then                Nodo( n' , izq , der )  
    else if n < n' then           busca( n , izq )  
    else if n > n' then           busca( n , der )
```

La implementación de la función “menor” en Lenguaje ML es la siguiente:

```
fun menor( Vacío )          =      Vacío |  
menor( Nodo( n , izq , der ) ) =  
    if izq = Vacío then        Nodo( n , izq , der )  
    else if izq ≠ Vacío       menor( izq )
```

La implementación de la función “mayor” en Lenguaje ML es la siguiente:

```
fun mayor( Vacío )          =      Vacío |  
mayor( Nodo( n , izq , der ) ) =  
    if der = Vacío then        Nodo( n , izq , der )  
    else if der ≠ Vacío       mayor( der )
```

La implementación de la función “borra” en Lenguaje ML es la siguiente:

fun borra(n , Vacío)	= Vacío
borra(n , Nodo(n' , izq , der))	=
if n = n' then	Vacío
if izq = Vacío & der = Vacío then	Nodo (mayor(izq) -> n' ,
else if izq ≠ Vacío & der = Vacío then	borra(mayor(izq) -> n' , mayor(izq)) ,
	der
)
else if izq = Vacío & der ≠ Vacío then	Nodo (menor(der) -> n' ,
	izq ,
	borra(menor(der) -> n' , menor(der))
)
else if izq ≠ Vacío & der ≠ Vacío then	Nodo (menor(der) -> n' ,
	izq ,
	borra(menor(der) -> n' , menor(der))
)
else if n < n' then	Nodo(n' , borra(n , izq) , der)
else if n > n' then	Nodo(n' , izq , borra(n , der))

Ahora veremos la implementación de estas funciones en el Lenguaje C:

```
// Archivo: arbol-binario-dos.c
#include <stdio.h>
#include <stdlib.h> // Para la función malloc. Funciones
// permiten asignar y liberar memoria dinámica

/* Declaración de la estructura Nodo para el arbol binario
 * Variables miembro:
 *     entero           elemento del nodo
 *     puntero a "struct Nodo"   sub-arbol izquierdo
 *     puntero a "struct Nodo"   sub-arbol derecho
 */
struct Nodo{
    int elemento;
    struct Nodo * izq, * der;
};

// prototipo de la función inserta
struct Nodo * inserta( int , struct Nodo * );

// prototipo de la función recorre
void recorre( struct Nodo * );
```

```
// Continuación del archivo: arbol-binario-dos.c
// prototipo de la función busca
struct Nodo * busca( int , struct Nodo * );

// prototipo de la función borra
struct Nodo * borra( int , struct Nodo * );

// prototipo de la función menor:
struct Nodo * menor( struct Nodo * );

// prototipo de la función mayor
struct Nodo * mayor( struct Nodo * );

/* Función: recorre
 * Destinada a mostrar en pantalla los elementos del arbol binario
 * en forma ordenada.
 * Retorna: void (nada)
 * Argumentos: puntero a "struct Nodo" a partir del cuál comenzará
 *              a mostrar sus elementos
 */
void recorre( struct Nodo * nodoElegido )
{
    if( nodoElegido != NULL )
    {
        recorre( nodoElegido->izq );
        printf( "\n %d \n" , nodoElegido -> elemento );
        recorre( nodoElegido->der );
    }
}
```

```
// Continuación del archivo: arbol-binario-dos.c
/* Función: inserta
 * Destinada a insertar un elemento al arbol a partir de un nodo
 * indicado mediante un "puntero a "struct Nodo"""
 * Retorna: un puntero a "struct Nodo"
 * Argumentos: un entero,    elemento que quiere insertar
 *              un puntero a "struct Nodo" a partir del cuál
 *              buscará el lugar adecuada para que al insertar
 *              el elemento, se mantenga el orden y no haya
 *              repetición de elementos.
 */
struct Nodo * inserta( int elemento , struct Nodo * nodoElegido )
{
    if( nodoElegido == NULL ){

        nodoElegido = malloc( sizeof( struct Nodo ) );

        /* malloc asigna sizeof( struct Nodo ) bytes y
         * devuelve un puntero a la memoria asignada.
         * La memoria no es borrada.
         */
        nodoElegido -> elemento = elemento;
        nodoElegido -> izq = nodoElegido -> der = NULL;
    }
    else if( elemento < nodoElegido -> elemento ){
        nodoElegido -> izq = inserta( elemento , nodoElegido -> izq );
        // inserta el elemento en el sub-arbol izquierdo
    }
    else if( elemento > nodoElegido -> elemento ){
        nodoElegido -> der = inserta( elemento , nodoElegido -> der );
        // inserta el elemento en el sub-arbol derecho
    }
    return nodoElegido;
}
```

```
// Continuación del archivo: arbol-binario-dos.c
/* Función: busca
 * Destinada a buscar un elemento en el arbol binario.
 * Retorna: un "puntero a "struct Nodo"" que es el que
 *           apunta al elemento que ha sido encontrado.
 *           NULL si el elemento no ha sido encontrado.
 * Argumentos: un entero     elemento buscado.
 *           un "puntero a "struct Nodo"" a partir del
 *           cuál comenzará a buscar.
 */
struct Nodo * busca( int elementoBuscado , struct Nodo * nodoElegido )
{
    if( nodoElegido != NULL )
    {
        if( elementoBuscado == nodoElegido -> elemento )
        {
            return nodoElegido;
        }
        else if( elementoBuscado < nodoElegido -> elemento )
        {
            return busca( elementoBuscado , nodoElegido -> izq );
        }
        else if( elementoBuscado > nodoElegido -> elemento )
        {
            return busca( elementoBuscado , nodoElegido -> der );
        }
    }
    return NULL;
}
```

```
/* Función: borra
 * Destinada a eliminar el elemento pasado como argumento.
 * Primero lo busca a partir del "punteo a "struct Nodo"" y
 * si lo encuentra, borra el Nodo en el que está, respetando
 * el orden de los elementos del arbol binario luego de que
 * se realiza la extracción.
 * Retorna:  un puntero a "struct Nodo"
 * Argumentos: un entero
 *           un puntero a "struct Nodo"
 */
```

```
// Continuación del archivo: arbol-binario-dos.c
struct Nodo * borra( int elementoAborrar , struct Nodo * nodoElegido )
{
    if( nodoElegido != NULL )
    {
        if( elementoAborrar == (nodoElegido -> elemento) )
        {
            if( (nodoElegido -> izq) == NULL && (nodoElegido -> der) == NULL )
            {
                return NULL;
            }
            else if( (nodoElegido -> izq) != NULL && (nodoElegido -> der) == NULL )
            {
                struct Nodo * mayorDELizq = mayor( nodoElegido -> izq );
                int numeroMayorDELizq = (mayorDELizq -> elemento);
                (nodoElegido -> elemento) = numeroMayorDELizq;
                (nodoElegido -> izq) = borra( numeroMayorDELizq , mayorDELizq );
                return nodoElegido;
            }
            else if( (nodoElegido -> izq) == NULL && (nodoElegido -> der) != NULL )
            {
                struct Nodo * menorDELder = menor( nodoElegido -> der );
                int numeroMenorDELder = (menorDELder -> elemento);
                (nodoElegido -> elemento) = numeroMenorDELder;
                (nodoElegido -> der) = borra( numeroMenorDELder , menorDELder );
                return nodoElegido;
            }
            else if( (nodoElegido -> izq) != NULL && (nodoElegido -> der) != NULL )
            {
                struct Nodo * menorDELder = menor( nodoElegido -> der );
                int numeroMenorDELder = (menorDELder -> elemento);
                (nodoElegido -> elemento) = numeroMenorDELder;
                (nodoElegido -> der) = borra( numeroMenorDELder , menorDELder );
                return nodoElegido;
            }
        }
        else if( elementoAborrar < (nodoElegido -> elemento) )
        {
            (nodoElegido -> izq) = borra( elementoAborrar , nodoElegido -> izq );
            return nodoElegido;
        }
        else if( elementoAborrar > (nodoElegido -> elemento) )
        {
            (nodoElegido -> der) = borra( elementoAborrar , nodoElegido -> der );
            return nodoElegido;
        }
    }
    return NULL;
}
```

```
// Continuación del archivo: arbol-binario-dos.c
/* Función: menor
 * Destinada a retornar el menor elemento que hay en el arbol
 * binario.
 * Retorna: puntero a "struct Nodo"
 * Argumentos: puntero a "struct Nodo"
 */
struct Nodo * menor( struct Nodo * nodoElegido )
{
    if( nodoElegido != NULL )
    {
        if( (nodoElegido -> izq) == NULL )
        {
            return nodoElegido;
        }
        else if( nodoElegido -> izq != NULL )
        {
            return menor( nodoElegido -> izq );
        }
    }
    return NULL;
}

/* Función: mayor
 * Destinada a retornar el mayor elemento que hay en el arbol
 * binario.
 * Retorna: un "puntero a "struct Nodo"""
 * Argumentos: un "puntero a "struct Nodo"""
 */
struct Nodo * mayor( struct Nodo * nodoElegido )
{
    if( nodoElegido != NULL )
    {
        if( (nodoElegido -> der) == NULL )
        {
            return nodoElegido;
        }
        else if( (nodoElegido -> der) != NULL )
        {
            return mayor( nodoElegido -> der );
        }
    }
    return NULL;
}
```

```
// Continuación del archivo: arbol-binario-dos.c

// Comienza el código ejecutable, función principal main
int main()
{
    struct Nodo * arbol = NULL;
    int indice;
    for( indice = 99 ; indice > 0 ; indice -= 2 )
    {
        arbol = inserta( indice , arbol );
    }
    struct Nodo * nodoDelElemento33 = busca( 33 , arbol );
    printf("El entero: %d “ , nodoDelElemento33->elemento );
    printf("se encuentra en: %p \n" , &(nodoDelElemento33->elemento));
    recorre( arbol );
    for( indice = 99 ; indice > 0 ; indice -= 2 )
    {
        struct Nodo * tmp = busca( indice , arbol );
        printf("elemento = %d “ , tmp->elemento );
        printf("en dirección = %p \n" , &(tmp->elemento) );
    }
    struct Nodo * minimo = menor( arbol );
    struct Nodo * maximo = mayor( arbol );
    printf("El elemento mínimo del arbol binario es: %d “ , minimo->elemento );
    printf("y está en: %p \n" , &(minimo->elemento) );
    printf("El elemento máximo del arbol binario es: %d “ , maximo->elemento );
    printf("y está en: %p \n" , &(maximo->elemento) );
    borra( 33 , arbol );
    //recorre( arbol );
    borra( 1 , arbol );
    borra( 99 , arbol );
    borra( 11 , arbol );
    borra( 55 , arbol );
    borra( 77 , arbol );
    recorre( arbol );
    minimo = menor( arbol );
    maximo = mayor( arbol );
    printf("El elemento mínimo del arbol binario es: %d “ , minimo->elemento );
    printf("y está en: %p \n" , &(minimo->elemento) );
    printf("El elemento máximo del arbol binario es: %d “ , maximo->elemento );
    printf("y está en: %p \n" , &(maximo->elemento) );
    return 0;      // salida exitosa del programa
}
// Fin del archivo: arbol-binario-dos.c
```

Se compila con:

gcc arbol-binario-dos.c -o arbol-binario-dos.out

Se ejecuta con:

./arbol-binario-dos.out

Nota: Como la salida en pantalla es muy extensa, se ha omitido.

Veamos **algunas cosas que podemos hacer con los arreglos.**

Por lo general, de un arreglo necesitamos saber:

- insertar elementos en el arreglo
- averiguar cuántos elementos contiene
- cuál es el mayor elemento
- cuál es el menor elemento
- si el arreglo contiene algún elemento en particular
- eliminar un elemento del arreglo. En este caso estamos hablando de reemplazar el valor que quiere borrar por un CERO.

Estas funciones están implementadas en el siguiente archivo de código fuente en Lenguaje C:

// Archivo: arreglos-y-funciones.c

```
#include <stdio.h>

#define SIZE 5      // Es una definición de una constante simbólica
                  // En adelante, cualquier aparición de SIZE será
                  // reemplazada por el texto de reemplazo
                  // Tamaño del arreglo

#define TRUE 0
#define FALSE 1

// prototipo de la función maximo
int * maximo ( int * );

// prototipo de la función minimo
int * minimo ( int * );

// prototipo de la función inserta
void inserta( int * );

// prototipo de la función existe
int existe( int , int * );

// prototipo de la función muestra
void muestra( int * );

// prototipo de la función busca
int * busca( int , int * );

// prototipo de la función elimina
void elimina( int , int * );

// prototipo de la función cuantos
int cuantos( int * );
```

// Continúa archivo: arreglos-y-funciones.c

```
/* Función: maximo
 * Destinada a retornar el máximo elemento del arreglo
 * Retorna: un puntero a entero
 * Argumentos: un puntero a entero a partir del cuál
 *               comenzará a buscar el máximo.
 */
int * maximo( int * conjunto )
{
    if( conjunto != NULL )
    {
        int indice , * max = &( conjunto[0] );
        for( indice = 0 ; indice < SIZE ; indice++ )
        {
            if( conjunto[ indice ] > (*max) )
            {
                max = &( conjunto[ indice ] );
                //indiceMax = indice;
            }
        }
        return max;
    }
    return NULL;
}
```

```
/* Función: minimo
 * Destinada a retornar el mínimo elemento del arreglo
 * Retorna: un puntero a entero
 * Argumentos: un puntero a entero a partir del cuál
 *               comenzará a buscar el mínimo.
 */
int * minimo( int * conjunto )
{
    if( conjunto != NULL )
    {
        int indice , * min = &( conjunto[0] );
        for( indice = 0 ; indice < SIZE ; indice++ )
        {
            if( conjunto[ indice ] < (*min) )
            {
                min = &( conjunto[ indice ] );
            }
        }
        return min;
    }
    return NULL;
}
```

// Continúa archivo: arreglos-y-funciones.c

```
/* Función: existe
 * Destinada a revisar si el entero que se le pasa como
 * argumento ya existe en el arreglo.
 * Argumentos: un entero el que se quiere verificar si
 *               existe.
 *               un puntero a entero a partir del cuál
 *               comenzará la búsqueda.
 * Retorna:    TRUE, es decir, 0 si ya existe el número
 *               FALSE, es decir, 1 si no está en el arreglo
 */
int existe( int num , int * conjunto )
{
    int resultado = FALSE;
    if( conjunto != NULL )
    {
        int indice;
        for( indice = 0 ; indice < SIZE ; indice++ )
        {
            if( num == conjunto[ indice ] )
            {
                resultado = TRUE;
            }
        }
        return resultado;
    }
    return resultado;
}
```

// Continúa archivo: arreglos-y-funciones.c

```
/* Función: inserta
 * Destinada a insertar elementos en el arreglo de manera
 * interactiva con el usuario, de manera que no permita
 * repetir elementos.
 * Retorna: void (nada).
 * Argumentos: un puntero entero que apunta al arreglo de
 *              enteros.
 */
void inserta( int * conjunto )
{
    if( conjunto != NULL )
    {
        int indice, tmp;
        for( indice = 0 ; indice < SIZE ; indice++ )
        {
            volviendoAintroducir:
            printf("Introduzca un entero para el elemento %d ° : " , indice + 1 );
            scanf( "%d" , &tmp );
            printf("Verificando si el entero ya existe en el arreglo, espere por favor !!!\n");
            if( existe( tmp , conjunto ) == FALSE )
            {
                conjunto[ indice ] = tmp;
            }
            else if( existe( tmp , conjunto ) == TRUE )
            {
                printf("El número que usted quiere ingresar ya existe en el arreglo!!!\n");
                printf("POR FAVOR INTRODUZCA OTRO NÚMERO DISTINTO!!!\n");
                goto volviendoAintroducir;
            }
        }
    }
}
```

// Continúa archivo: arreglos-y-funciones.c

```
/* Función: muestra
 * Destinada a mostrar en pantalla el contenido del arreglo.
 * Retorna: void (nada)
 * Argumentos: un puntero a entero a partir del cuál comienza
 *              mostrar el arreglo.
 */
void muestra( int * conjunto )
{
    if( conjunto != NULL )
    {
        int indice;
        printf("\n");
        for( indice = 0 ; indice < SIZE ; indice++ )
        {
            printf("conjunto[ %d ] = %d ", indice , conjunto[indice] );
            printf(" ubicado en: %p \n" , &( conjunto[indice] ) );
        }
        printf("\n");
    }
}

/* Función: busca
 * Destinada a retornar el puntero a entero donde se ubica
 * el elemento que se está buscando.
 * Retorna: un puntero a entero que es el que contiene el
 *          número que se está buscando
 *          NULL si el elemento no está en el arreglo
 * Argumentos: un entero      que se está buscando
 *              un puntero a entero a partir del cuál
 *              comienza la búsqueda
 */
int * busca( int numero , int * conjunto )
{
    if( conjunto != NULL )
    {
        int indice;
        for( indice = 0 ; indice < SIZE ; indice++ )
        {
            if( numero == conjunto[ indice ] )
            {
                return ( &( conjunto[ indice ] ) );
            }
        }
    }
    return NULL;
}
```

// Continúa archivo: arreglos-y-funciones.c

```
/* Función: elimina
 * Destinada a “borrar” el entero pasado como argumento, si es que
 * está dentro del arreglo su valor es reemplazado por CERO.
 * Retorna: void (nada)
 * Argumentos: un entero    el que se quiere eliminar.
 *              un puntero a entero   A partir del cuál
 *                                comienza a buscar.
 */
void elimina( int numero , int * conjunto )
{
    if( conjunto != NULL )
    {
        if( existe( numero , conjunto ) == TRUE )
        {
            int indice;
            for( indice = 0 ; indice < SIZE ; indice++ )
            {
                if( numero == conjunto[ indice ] )
                {
                    conjunto[ indice ] = 0;
                }
            }
        }
        else
        {
            printf("El elemento que quiere PONER EN CERO no se encuentra en el arreglo!!!\n");
        }
    }
}
/* Función: cuantos
 * Destinada a retornar la cantidad de elementos que posee el arreglo.
 * Retorna: un entero
 * Argumentos: un puntero a int a partir del cuál empieza a buscar.
 */
int cuantos( int * conjunto )
{
    int cantidad = 0;
    if( conjunto != NULL )
    {
        int indice;
        for( indice = 0 ; indice < SIZE ; indice++ )
        {
            cantidad++;
        }
        return cantidad;
    }
    return cantidad;
}
```

```
// Continúa archivo: arreglos-y-funciones.c

// comienza el código ejecutable principal

int main()
{
    int arreglo[ SIZE ];
    inserta( arreglo );           // Esta es una llamada a una función que va a operar en forma interactiva
    int * mayor = maximo( arreglo );
    int * menor = minimo( arreglo );
    printf("El elemento mayor es: %d \n" , *mayor );
    printf("El elemento menor es: %d \n" , *menor );
    int * buscadoA = busca( 3 , arreglo );
    printf("Buscando elemento 3 en el arreglo: \n" );
    if( buscadoA != NULL )
    {
        printf("El elemento %d ha sido encontrado en: %p \n" , *buscadoA , buscadoA );
    }
    else
    {
        printf("El elemento 3 no ha sido encontrado en el arreglo!!!\n");
    }
    muestra( arreglo );
    elimina( 2 , arreglo );
    muestra( arreglo );
    elimina( 1 , arreglo );
    int * mayorDos = maximo( arreglo );
    int * menorDos = minimo( arreglo );
    printf("El elemento mayor es: %d \n" , *mayorDos );
    printf("El elemento menor es: %d \n" , *menorDos );
    muestra( arreglo );
    int cantidad = cuantos( arreglo );
    printf("La cantidad de elementos del arreglo es: %d \n" , cantidad );
    return 0;      // salida exitosa del programa
}
// Fin del archivo: arreglos-y-funciones.c
```

Se compila con: **gcc arreglos-y-funciones.c -o arreglos-y-funciones.out**

Se ejecuta con: **./arreglos-y-funciones.out**

Nota: la salida en pantalla no se muestra porque depende de cómo se ha realizado la inserción de elementos en el arreglo.

Para entender mejor algunas líneas de este código primero es necesario introducir nuevos conceptos.

Definición de constantes simbólicas:

Es una **mala práctica** poner “**números mágicos**” en un programa, ya que proporcionan muy poca información a quién tenga que leer el programa, y son difíciles de modificar en una forma sistemática. Una manera de tratar a esos números mágicos es darles nombres significativos. Una línea **#define** define un **nombre simbólico o constante simbólica** como una cadena de caracteres especial:

#define nombre texto de reemplazo;

A partir de esto, cualquier ocurrencia de **nombre** (que no esté entre comillas ni como parte de otro nombre) se sustituirá por el **texto de reemplazo** correspondiente. El **nombre** tiene la misma forma que un nombre de variable: una secuencia de letras y dígitos que comienza con una letra. El **texto de reemplazo** puede ser cualquier secuencia de caracteres; no está limitado a números.

Los nombres de las constantes simbólicas, por convención, se escriben con letras mayúsculas, de modo que se puedan distinguir fácilmente de los nombres de las variables escritos en minúsculas.

Ejemplo de líneas **#define** que se utilizaron en el archivo **arreglos-y-funciones.c** son los siguientes:

- **#define SIZE 5**
- **#define TRUE 0**
- **#define FALSE 1**

Tanto las sentencias **#define** así como **#include**, entre otras que veremos más adelante, son órdenes interpretadas por el:

PREPROCESADOR DE C

C proporciona ciertas facilidades del lenguaje por medio de un **preprocesador**, que **conceptualmente es un primer paso separado en la compilación**. Los dos elementos que se usan con más frecuencia son:

- **#include** para incluir el contenido de un archivo durante la compilación
- **#define** para reemplazar un símbolo por una secuencia arbitraria de caracteres

Inclusión de archivos:

La inclusión de archivos facilita el manejo de grupos de **#define** y declaraciones, entre otras cosas. Un ejemplo claro de esto es la típica línea: **#include <stdio.h>**, éste es un archivo que **reúne una serie de declaraciones y definiciones**, es por esto que se lo denomina “**header**” o “**archivo de encabezados**”. Cualquier línea de código fuente que tenga la siguiente forma, se reemplaza por el contenido del archivo nombre:

- **#include “nombre”** Si el **nombre** del archivo se encierra entre comillas, la **búsqueda** del archivo **comienza** normalmente **donde se encontró el archivo de código fuente del programa**, si no se encuentra allí **continúa** la búsqueda en una **ruta predefinida por la implantación del S.O**
- **#include <nombre>** Si el **nombre** del archivo se **delimita por < y >**, la **búsqueda** sigue **una ruta predefinida por la implantación del S.O**

El uso de **#include** es la mejor manera de enlazar las declaraciones para un programa grande.

Garantiza que todos los archivos fuente se suplirán con las mismas definiciones y declaraciones de variables, y así elimina un tipo de error particularmente desagradable. Por supuesto, cuando se cambia un archivo que es referenciado en otros archivos mediante **#include**, se deben recomilar todos los archivos que dependen de éste.

Algunas macros predefinidas por el compilador:

Identificadores de compilador:

- __TURBO_C__ ó __BORLAND__ si está usando TURBO C/C++
- __GNUC__ si está usando el Gnu C Compiler
- __FILE__ es el nombre del archivo
- __DATE__ es la fecha en que se realizó la compilación
- __HOUR__ ó __TIME__ es la hora en que se realizó la compilación
- __LINE__ es la línea del archivo de código fuente
- __FUNCTION__ Sólo en GCC, es el nombre de la función

Veamos un ejemplo que emplea estas macros:

```
// Archivo: macros-predefinidas.c
#include <stdio.h>
```

```
int main()
{
    printf("\n_____________________________________\n");
    printf("Programa: %s \n", __FILE__);
    printf("compilado el: %s a las %s \n", __DATE__, __TIME__);
    printf("Esto está en la línea N°: %d \n", __LINE__);
    printf("y esto está en la línea N°: %d \n", __LINE__);
    printf("La función es: %s \n", __FUNCTION__);
    printf("_____________________________________\n");
    return 0; // salida exitosa del programa
}
// Fin del archivo: macros-predefinidas.c
```

Se compila con: **gcc -Wall macros-predefinidas.c -o macros-predefinidas.out**

Se ejecuta con: **./macros-predefinidas.out**

Substitución de macros:

Una definición de la forma:

#define nombre texto de reemplazo

pide la sustitución de una **macro** del tipo más sencillo, por lo que las siguientes ocurrencias de **nombre** serán sustituidas por el **texto de reemplazo**. Normalmente el texto de reemplazo es el resto de la línea, pero una **definición extensa** puede continuarse en varias líneas, colocando una **** al final de cada línea que va a continuar. El alcance de un nombre definido con **#define** va desde su punto de definición hasta el fin del archivo fuente que se compila.

Cualquier **nombre** puede definirse con cualquier **texto de reemplazo**, aunque las sustituciones se realizan sólo para elementos, y no sucede dentro de cadenas delimitadas por comillas.

Las **líneas #define** no solo sirven para definir constantes simbólicas, pues también se **pueden definir macros**. Por ejemplo:

```
#define      porsiempre      for( ; ; )
```

Define una nueva palabra “**porsiempre**” que se sustituirá por un ciclo infinito.

También es posible definir macros con argumentos, para que el texto de reemplazo pueda ser diferente en distintas llamadas a la macro. Por ejemplo:

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

Aparenta ser una llamada a función pero en realidad el uso de **max(A , B)** se expande a código. Cada ocurrencia de un parámetro formal **A** o **B** será reemplazada por el argumento real correspondiente que tiene en ese momento, al hacer la llamada. Mientras los argumentos se traten consistentemente, esta macro servirá para cualquier tipo de datos; no hay necesidad de diferentes tipos de **max(A , B)** para diferentes tipos de datos, como lo habría con las funciones. Esta forma de definir la macro tiene algunos riesgos, pues las expresiones se evalúan dos veces y es malo si involucra efectos colaterales como operadores incrementales o de entrada y salida. Es el caso, por ejemplo de una invocación a la macro de la siguiente manera:

max(i++ , j++) ya que incrementaría el valor más grande dos veces.

En una declaración de una macro los **parámetros formales** no se reemplazan si están dentro de cadenas entre comillas. Sin embargo, si un nombre de parámetro está precedido por un # en el **texto de reemplazo**, esta combinación, **#nombre-de-parámetro**, se expandirá en una cadena entre comillas, con el parámetro reemplazado por el argumento real con el que se está invocando en ese momento.

Veamos un ejemplo, la **declaración de la macro**:

#define dprintf(expr) printf(#expr “ = %g\n” , expr)

cuando se invoca la macro, como en:

dprintf(x/y);

la macro se expande en:

printf(“x/y” “ = %g\n” , x/y);

y las cadenas se concatenan quedando así:

printf(“x/y = %g\n” , x/y);

Si el argumento real incluye caracteres como:

- “ entonces al expandir se lo reemplaza por \”
 - \ entonces al expandir se lo reemplaza por \n
- de esta manera se obtiene una constante de cadena legítima.

Más aún, el **preprocesador de C** posee el **operador ##** que proporciona una **forma de concatenar argumentos reales durante la expansión de una macro**. Si un **parámetro** que está en el **texto de reemplazo** es adyacente a un **##**, éste es reemplazado por el argumento real, se eliminan el **##** y los espacios en blanco que lo rodean, y el resultado se rastrea de nuevo. Por ejemplo:

la macro paste: **#define paste(front , back) front ## back**

cuando se invoca con: **paste(nombre , 1)**

se expandirá como: **nombre1**

Así como se puede realizar una definición, también se puede **revocar una definición** mediante una línea **#undef**, de este modo, se consigue que la definición deje de tener efecto. Es particularmente útil para asegurar que una llamada a una rutina es realmente una invocación a una función, y no una llamada a una macro. **#undef** es otra sentencia reconocida por el preprocesador de C.

El preprocesador de C también permite realizar:

Inclusión condicional de archivos:

Es posible **controlar el preprocesamiento** mismo con **proposiciones condicionales** que se evalúan durante esta etapa. Esto proporciona **una forma de incluir código selectivamente**, dependiendo del **valor de las condiciones evaluadas durante la compilación**.

Una línea **#if** evalúa una **expresión constante entera**

(que no puede incluir sizeof, casts, o constantes enum)

Si la expresión resulta:

- **diferente de CERO:** se incluyen las siguientes líneas hasta que aparezca un **#endif** , **#elif** (que es similar a un else if) , o un **#else**
- **CERO:** entonces se omiten las siguientes líneas hasta que aparezca un **#endif** , **#elif**, o un **#else**

Dentro del contenido de una línea **#if** suele utilizarse la expresión **defined(nombre)** que al ser evaluada devuelve:

- **1(UNO):** si el nombre ya ha sido definido
- **0(CERO):** si el nombre no ha sido definido

Por ejemplo:

```
#if !defined( HDR )
```

```
#define HDR
```

```
// aquí va el contenido del archivo: hdr.h
```

```
#endif
```

Esto asegura que el contenido del archivo “hdr.h” se incluya sólo una vez.

Se recomienda usar esta forma incluir archivos en forma consistente, pues si hace de esta manera cada “header” puede en sí mismo incluir cualquier otro del que dependa, evitando que el programador tenga que tratar con esta interdependencia.

Veamos otro ejemplo:

```
#if SYSTEM == SYSV
    #define HDR "sysv.h"
#elif SYSTEM == BSD
    #define HDR "bsd.h"
#elif SYSTEM == MSDOS
    #define HDR "msdos.h"
#else
    #define HDR "default.h"
#endif
#include HDR
```

Hay dos líneas más, **#ifdef** y **#ifndef**, son formas especializadas que prueban si un nombre está definido (**#ifdef** verifica si el nombre está definido, **#ifndef** chequea si el nombre no está definido), veamos un ejemplo:

```
#ifndef HDR
#define HDR
/* contenido de HDR va aquí */
#endif //Es el mismo ejemplo que habíamos visto antes, pero haciendo uso de #ifndef
```

Typedef:

C provee una facilidad llamada typedef para **crear “nuevos tipos de datos”, en realidad nuevas formas de identificar a tipos de datos conocidos.**

Por ejemplo la declaración:

```
typedef int i; // Hace a “i” un sinónimo de “int”
```

¿Qué significa sinónimo?

Pues de ahora en adelante podremos escribir: i j, k; // j es un entero. k es un entero.

Otros ejemplos:

- `typedef double * pd` // pd es un “**puntero a double**” y podemos escribir:
`pd p1, p2, p3;` // p1 , p2 y p3 son variables del tipo “puntero a double”

- `typedef struct Nodo{`
 `int elemento;`
 `struct Nodo * izq, *der;`
`} NodoBinario;` /* **NodoBinario** es sinónimo de
* “**struct Nodo{ int elemento; struct Nodo * izq, *der; }**”
* De ahora en adelante puede utilizar **NodoBinario**, en
* lugar de “**struct Nodo**” para las **declaraciones de nodos**
* de un **arbol binario**.
*/

- `typedef struct Nodo * PNodoBinario` /* **PNodoBinario** es sinónimo de “**struct Nodo ***”
* PNodoBinario es equivalente a “puntero a struct
* nodo”.
*/

Se debe destacar que **una declaración typedef no crea un nuevo tipo de dato en ningún sentido; simplemente agrega un nuevo nombre para algún tipo de dato existente**. En efecto typedef funciona en forma similar al #define, excepto que al ser interpretado por el compilador de C, puede realizar sustituciones textuales que están más allá de las capacidades del preprocesador.

Hay dos razones principales para emplear typedef. La primera es parametrizar un programa contra los problemas de transportabilidad, pues al usar typedef para aquellos tipos de datos que pueden ser dependientes de la máquina, cuando se traslada el código hacia otra máquina diferente, solo hay que modificar los typedef. La segunda razón es proporcionar mejor documentación para un programa, al emplear nombres significativos del dominio del problema que el programa está resolviendo.

Ahora veamos un ejemplo de una declaración complicada:

```
double (* pf ) ( double ); // pf es un “puntero a una función” que toma un double y
                           // devuelve un double
```

Por ejemplo en el siguiente código:

```
// Archivo: puntero-a-funciones.c
#include <stdio.h>
#include <math.h>    // Para poder usar funciones matemáticas, trigonométricas
#include <stdlib.h>
int main()
{
    double valor = 0.0 , valor2 = 3.1416 , valor3 = valor2/2 , valor4 = valor2/4;
    double (* punteroAfuncion ) ( double );
    // Se declara un puntero a una función que toma un double como argumento
    // y retorna un double

    printf("\n");
    punteroAfuncion = sin; // Ahora apunta a la función sin
    printf(" sen( %f ) = %f \n" , valor , punteroAfuncion( valor ) );
    printf(" sen( %f ) = %f \n" , valor2 , punteroAfuncion( valor2 ) );
    printf(" sen( %f ) = %f \n" , valor3 , punteroAfuncion( valor3 ) );
    printf(" sen( %f ) = %f \n" , valor4 , punteroAfuncion( valor4 ) );
    printf("\n");
    punteroAfuncion = exp; // Ahora apunta a la función exp
    printf(" exp( %f ) = %f \n" , valor , punteroAfuncion( valor ) );
    printf(" exp( %f ) = %f \n" , valor2 , punteroAfuncion( valor2 ) );
    printf(" exp( %f ) = %f \n" , valor3 , punteroAfuncion( valor3 ) );
    printf(" exp( %f ) = %f \n" , valor4 , punteroAfuncion( valor4 ) );
    printf("\n");
    punteroAfuncion = cos; // Ahora apunta a la función cos
    printf(" cos( %f ) = %f \n" , valor , punteroAfuncion( valor ) );
    printf(" cos( %f ) = %f \n" , valor2 , punteroAfuncion( valor2 ) );
    printf(" cos( %f ) = %f \n" , valor3 , punteroAfuncion( valor3 ) );
    printf(" cos( %f ) = %f \n" , valor4 , punteroAfuncion( valor4 ) );
    printf("\n");

    return 0; // salida exitosa del programa
}
```

//Fin del Archivo: puntero-a-funciones.c

Para poder compilar este archivo se debe usar el gcc junto con el argumento para línea de comandos **-lm** de la siguiente manera:

```
gcc -Wall puntero-a-funciones.c -o puntero-a-funciones.out -lm
```

Se ejecuta con: **./puntero-a-funciones.out**
y su salida en pantalla es:

```
sen( 0.000000 ) = 0.000000
sen( 3.141600 ) = -0.000007
sen( 1.570800 ) = 1.000000
sen( 0.785400 ) = 0.707108
```

```
exp( 0.000000 ) = 1.000000
exp( 3.141600 ) = 23.140863
exp( 1.570800 ) = 4.810495
exp( 0.785400 ) = 2.193284
```

```
cos( 0.000000 ) = 1.000000
cos( 3.141600 ) = -1.000000
cos( 1.570800 ) = -0.000004
cos( 0.785400 ) = 0.707105
```

Es muy útil saber que como estamos usando un **Sistema Operativo** licenciado conforme a la **GPL (General Public License o licencia pública general)** tenemos al alcance de la mano mucha **información**, solo es necesario tener los **paquetes de la documentación debidamente instalados**, para poder hacer uso por ejemplo de la **utilidad para la consola: “man”**
Veamos que nos puede brindar, al consultar la página del manual referida a una función muy conocida “qsort”, sobretodo nos interesa la siguiente porción:

SINOPSIS

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

```
void qsort( void *base , size_t nmiemb , size_t tam , int (*compar)(const void *, const void *) );
```

En la sección titulada “SINOPSIS” podemos observar en qué archivo está definida esta función y también el prototipo de la función qsort. Vamos a desglosar el prototipo de qsort para entenderlo mejor:

- void La función qsort no devuelve ningún valor de retorno
- void *base Es un “**puntero a void**”, se trata de la **declaración más genérica de un puntero**, de esta manera **un “puntero a void”** puede ser **valorizado con valores correspondientes a “punteros a cualquier tipo de dato”**, o sea, una manera de dar una **definición estandar de un puntero que luego se puede reutilizar con el valor adecuado**. En este caso servirá para pasar como parámetro el **puntero al primer elemento del arreglo que se quiere ordenar en forma ascendente** mediante este **método de ordenación rápida**.
- size_t nmiemb Aquí **size_t** es en realidad **un sinónimo** creado con:
typedef unsigned int size_t
es decir que estamos hablando de la declaración: **unsigned int nmiemb**
En este caso **nmiemb** se refiere al **número de miembros que posee el arreglo**

- `size_t tam` En este caso, con **tam** significa el **tamaño que ocupa en memoria un elemento del arreglo**
- `int (*compar)(const void *, const void *)` Se trata de un “**puntero a una función**” de **comparación** que **toma como argumentos a dos “punteros a constantes void”**, que apuntan a los objetos a comparar y **retorna un entero que será menor que, igual a, o mayor que cero si el primer argumento se considera respectivamente menor, igual o mayor que el segundo**. Si los dos miembros se **comparan como iguales**, su **orden en el vector clasificado queda indefinido**.

Con esta información podemos comenzar a utilizar la función qsort como se muestra a continuación:

```
// Archivo: qsort.c
#include <stdio.h>
#include <stdlib.h>

#define TAMANIOdeARREGLO 10
#define TAMANIOdeELEMENTO     sizeof( a[0] )

int a[]={10,1,9,2,8,3,7,4,6,5};

int compara( const void * parametro1 , const void * parametro2 )
{
    return ( *( (int *) parametro1 ) - *( (int *) parametro2 ) );
}

int main()
{
    int indice;

    printf("\n");
    for( indice = 0 ; indice < TAMANIOdeARREGLO ; indice++ )
    {
        printf("a[ %d ] = %d en %p \n" , indice , a[indice] , &(a[indice]) );
    }
    qsort( a , TAMANIOdeARREGLO , TAMANIOdeELEMENTO , compara );
    // leer: man qsort
    printf("\n");
    for( indice = 0 ; indice < TAMANIOdeARREGLO ; indice++ )
    {
        printf("a[ %d ] = %d en %p \n" , indice , a[indice] , &(a[indice]) );
    }
    printf("\n");
    return 0;      // salida exitosa del programa
}
// Fin del archivo: qsort.c
```

Se compila con:

gcc -Wall ./qsort.c -o qsort.out

Se ejecuta con: **./qsort.out**

Y su salida en pantalla es:

```
a[ 0 ] = 10 en 0x8049620
a[ 1 ] = 1 en 0x8049624
a[ 2 ] = 9 en 0x8049628
a[ 3 ] = 2 en 0x804962c
a[ 4 ] = 8 en 0x8049630
a[ 5 ] = 3 en 0x8049634
a[ 6 ] = 7 en 0x8049638
a[ 7 ] = 4 en 0x804963c
a[ 8 ] = 6 en 0x8049640
a[ 9 ] = 5 en 0x8049644
```

```
a[ 0 ] = 1 en 0x8049620
a[ 1 ] = 2 en 0x8049624
a[ 2 ] = 3 en 0x8049628
a[ 3 ] = 4 en 0x804962c
a[ 4 ] = 5 en 0x8049630
a[ 5 ] = 6 en 0x8049634
a[ 6 ] = 7 en 0x8049638
a[ 7 ] = 8 en 0x804963c
a[ 8 ] = 9 en 0x8049640
a[ 9 ] = 10 en 0x8049644
```

Con una leve modificación en el código fuente anterior, se puede lograr que la función de comparación ordene el arreglo en forma descendente:

```
int compara( const void * parametro1 , const void * parametro2 )
{
    return ( *( (int *) parametro2 ) - *( (int *) parametro1 ) );
}
```

Ahora que sabemos cómo emplear **typedef** podemos escribir **otra versión del código fuente para implementar árboles binarios**:

```
// Archivo: arbol-binario-tres.c
#include <stdio.h>
#include <stdlib.h> // Para la función malloc. Funciones que
// permiten asignar y liberar memoria dinámica

/* Declaración de un sinónimo de "puntero a estructura Nodo"
 * para el arbol binario
 * Variables miembro de la estructura Nodo:
 *     entero           elemento del nodo
 *     puntero a "struct Nodo"   sub-arbol izquierdo
 *     puntero a "struct Nodo"   sub-arbol derecho
 */
typedef struct Nodo{ int elemento; struct Nodo * izq, * der;} * PNodo ;

// prototipo de la función inserta
PNodo inserta( int , PNodo );
```

```
// Continúa el archivo: arbol-binario-tres.c
// prototipo de la función recorre
void recorre( PNodo );

// prototipo de la función busca
PNodo busca( int , PNodo );

// prototipo de la función borra
PNodo borra( int , PNodo );

// prototipo de la función menor:
PNodo menor( PNodo );

// prototipo de la función mayor
PNodo mayor( struct Nodo * );

/* Función: recorre
 * Destinada a mostrar en pantalla los elementos del arbol binario
 * en forma ordenada.
 * Retorna: void (nada)
 * Argumentos: puntero a "struct Nodo" a partir del cuál comenzará
 *               a mostrar sus elementos
 */
void recorre( PNodo nodoElegido )
{
    if( nodoElegido != NULL )
    {
        recorre( nodoElegido->izq );
        printf( "\n %d \n" , nodoElegido -> elemento );
        recorre( nodoElegido->der );
    }
}
```

```
// Continúa el archivo: arbol-binario-tres.c

/* Función: inserta
 * Destinada a insertar un elemento al arbol a partir de un nodo
 * indicado mediante un "puntero a "struct Nodo"""
 * Retorna: un puntero a "struct Nodo"
 * Argumentos: un entero,    elemento que quiere insertar
 *              un puntero a "struct Nodo" a partir del cuál
 *              buscará el lugar adecuada para que al insertar
 *              el elemento, se mantenga el orden y no haya
 *              repetición de elementos.
 */

PNodo inserta( int elemento , PNodo nodoElegido )
{
    if( nodoElegido == NULL ){

        nodoElegido = malloc( sizeof( struct Nodo ) );

        /* malloc asigna sizeof( struct Nodo ) bytes y
         * devuelve un puntero a la memoria asignada.
         * La memoria no es borrada.
         */
        nodoElegido -> elemento = elemento;
        nodoElegido -> izq = nodoElegido -> der = NULL;
    }
    else if( elemento < nodoElegido -> elemento ){
        nodoElegido -> izq = inserta( elemento , nodoElegido -> izq );
        // inserta el elemento en el sub-arbol izquierdo
    }
    else if( elemento > nodoElegido -> elemento ){
        nodoElegido -> der = inserta( elemento , nodoElegido -> der );
        // inserta el elemento en el sub-arbol derecho
    }
    return nodoElegido;
}
```

// Continúa el archivo: arbol-binario-tres.c

```
/* Función: busca
 * Destinada a buscar un elemento en el arbol binario.
 * Retorna: un "puntero a "struct Nodo"" que es el que
 *           apunta al elemento que ha sido encontrado.
 *           NULL si el elemento no ha sido encontrado.
 * Argumentos: un entero     elemento buscado.
 *             un "puntero a "struct Nodo"" a partir del
 *             cuál comenzará a buscar.
 */
PNodo busca( int elementoBuscado , PNodo nodoElegido )
{
    if( nodoElegido != NULL )
    {
        if( elementoBuscado == nodoElegido -> elemento )
        {
            return nodoElegido;
        }
        else if( elementoBuscado < nodoElegido -> elemento )
        {
            return busca( elementoBuscado , nodoElegido -> izq );
        }
        else if( elementoBuscado > nodoElegido -> elemento )
        {
            return busca( elementoBuscado , nodoElegido -> der );
        }
    }
    return NULL;
}
```

/* Función: borra

```
* Destinada a eliminar el elemento pasado como argumento. Primero lo busca a partir del
* "punteo a "struct Nodo"" y si lo encuentra, borra el Nodo en el que está, respetando
* el orden de los elementos del arbol binario luego de que se realiza la extracción.
* Retorna:  un puntero a "struct Nodo"
* Argumentos:      un entero
*                 un puntero a "struct Nodo"
*/
```

```
// Continúa el archivo: arbol-binario-tres.c
PNodo borra( int elementoAborrar , PNodo nodoElegido )
{
    if( nodoElegido != NULL )
    {
        if( elementoAborrar == (nodoElegido -> elemento) )
        {
            if( (nodoElegido -> izq) == NULL && (nodoElegido -> der) == NULL )
            {
                return NULL;
            }
            else if( (nodoElegido -> izq) != NULL && (nodoElegido -> der) == NULL )
            {
                PNodo mayorDELizq = mayor( nodoElegido -> izq );
                int numeroMayorDELizq = (mayorDELizq -> elemento);
                (nodoElegido -> elemento) = numeroMayorDELizq;
                (nodoElegido -> izq) = borra( numeroMayorDELizq , mayorDELizq );
                return nodoElegido;
            }
            else if( (nodoElegido -> izq) == NULL && (nodoElegido -> der) != NULL )
            {
                PNodo menorDELder = menor( nodoElegido -> der );
                int numeroMenorDELder = (menorDELder -> elemento);
                (nodoElegido -> elemento) = numeroMenorDELder;
                (nodoElegido -> der) = borra( numeroMenorDELder , menorDELder );
                return nodoElegido;
            }
            else if( (nodoElegido -> izq) != NULL && (nodoElegido -> der) != NULL )
            {
                PNodo menorDELder = menor( nodoElegido -> der );
                int numeroMenorDELder = (menorDELder -> elemento);
                (nodoElegido -> elemento) = numeroMenorDELder;
                (nodoElegido -> der) = borra( numeroMenorDELder , menorDELder );
                return nodoElegido;
            }
        }
        else if( elementoAborrar < (nodoElegido -> elemento) )
        {
            (nodoElegido -> izq) = borra( elementoAborrar , nodoElegido -> izq );
            return nodoElegido;
        }
        else if( elementoAborrar > (nodoElegido -> elemento) )
        {
            (nodoElegido -> der) = borra( elementoAborrar , nodoElegido -> der );
            return nodoElegido;
        }
    }
    return NULL;
}
```

```
// Continúa el archivo: arbol-binario-tres.c
/* Función: menor
 * Destinada a retornar el menor elemento que hay en el arbol
 * binario.
 * Retorna: puntero a "struct Nodo"
 * Argumentos: puntero a "struct Nodo"
 */
PNodo menor( PNodo nodoElegido )
{
    if( nodoElegido != NULL )
    {
        if( (nodoElegido -> izq) == NULL )
        {
            return nodoElegido;
        }
        else if( nodoElegido -> izq != NULL )
        {
            return menor( nodoElegido -> izq );
        }
    }
    return NULL;
}

/* Función: mayor
 * Destinada a retornar el mayor elemento que hay en el arbol
 * binario.
 * Retorna: un "puntero a "struct Nodo"""
 * Argumentos: un "puntero a "struct Nodo"""
 */
PNodo mayor( PNodo nodoElegido )
{
    if( nodoElegido != NULL )
    {
        if( (nodoElegido -> der) == NULL )
        {
            return nodoElegido;
        }
        else if( (nodoElegido -> der) != NULL )
        {
            return mayor( nodoElegido -> der );
        }
    }
    return NULL;
}
```

```
// Continúa el archivo: arbol-binario-tres.c
// Comienza el código ejecutable, función principal main
int main()
{
    PNodo arbol = NULL;
    int indice;
    for( indice = 99 ; indice > 0 ; indice -= 2 )
    {
        arbol = inserta( indice , arbol );
    }
    PNodo nodoDelElemento33 = busca( 33 , arbol );
    printf("El entero: %d “ , nodoDelElemento33->elemento );
    printf("se encuentra en: %p \n" , &(nodoDelElemento33->elemento) );
    recorre( arbol );
    for( indice = 99 ; indice > 0 ; indice -= 2 )
    {
        PNodo tmp = busca( indice , arbol );
        printf("elemento = %d “ , tmp->elemento );
        printf("en dirección = %p \n" , &(tmp->elemento) );
    }
    PNodo minimo = menor( arbol );
    PNodo maximo = mayor( arbol );
    printf("El elemento mínimo del arbol binario es: %d “ , minimo->elemento );
    printf("y está en: %p \n" , &(minimo->elemento) );
    printf("El elemento máximo del arbol binario es: %d “ , maximo->elemento );
    printf("y está en: %p \n" , &(maximo->elemento) );
    borra( 33 , arbol );
    //recorre( arbol );
    borra( 1 , arbol );
    borra( 99 , arbol );
    borra( 11 , arbol );
    borra( 55 , arbol );
    borra( 77 , arbol );
    recorre( arbol );
    minimo = menor( arbol );
    maximo = mayor( arbol );
    printf("El elemento mínimo del arbol binario es: %d “ , minimo->elemento );
    printf("y está en: %p \n" , &(minimo->elemento) );
    printf("El elemento máximo del arbol binario es: %d “ , maximo->elemento );
    printf("y está en: %p \n" , &(maximo->elemento) );
    return 0;      // salida exitosa del programa
}
// Fin del archivo: arbol-binario-tres.c
```

Se compila con: **gcc -Wall arbol-binario-tres.c -o arbol-binario-tres.out**

Y se ejecuta con: **./arbol-binario-tres.out**

Cadenas de caracteres en C: Arreglos de caracteres

El Lenguaje de Programación C no posee el tipo de dato “string”.

Se puede lograr algo similar a un string mediante un arreglo de caracteres.

Se puede usar esta notación:

```
char a[] = {'h','o','l','a','\0'};           // Para declarar la cadena de caracteres "hola" como un
                                              // arreglo de caracteres
                                              // Debe finalizar con el carácter nulo: '\0'

printf( "%s \n ", a );                     // %s Para mostrar en pantalla la cadena de caracteres
                                              // referenciada por el nombre del arreglo, que no es otra cosa que
                                              // un "puntero constante al primer elemento del arreglo de
                                              // caracteres"
```

Una notación equivalente es:

```
char a[] = "hola";           // El compilador de C lo interpreta de la misma manera, es una
                                              // alineación de la notación anterior, la cantidad de caracteres miembros
                                              // de este arreglo sigue siendo 5 y el último elemento continúa siendo el
                                              // carácter nulo: '\0'
```

Obviamente, también es correcto escribir:

```
char * a = "hola";           // Declara a la cadena de caracteres, cuyo nombre "a" es un "puntero
                                              // a un carácter" que apunta al primer carácter de la cadena: 'h'. La
                                              // finalización de la cadena se sigue indicando con el carácter nulo: '\0'
```

Podemos ver un ejemplo con el siguiente código fuente:

```
// Archivo: string.c
#include <stdio.h>
int main()
{
    //char a[] = {'h','o','l','a','\0'};
    //char a[] = "hola";
    char * a = "hola";
    printf("%s \n ", a);
    return 0;      // salida exitosa del programa
}
```

// Fin del archivo string.c

que se compila con: **gcc -Wall string.c -o string.out**

y se ejecuta con: **./string.out**

Arreglos multidimensionales:

Con los arreglos multidimensionales se **rompe la simetría** que habíamos visto **entre arreglos y punteros**.

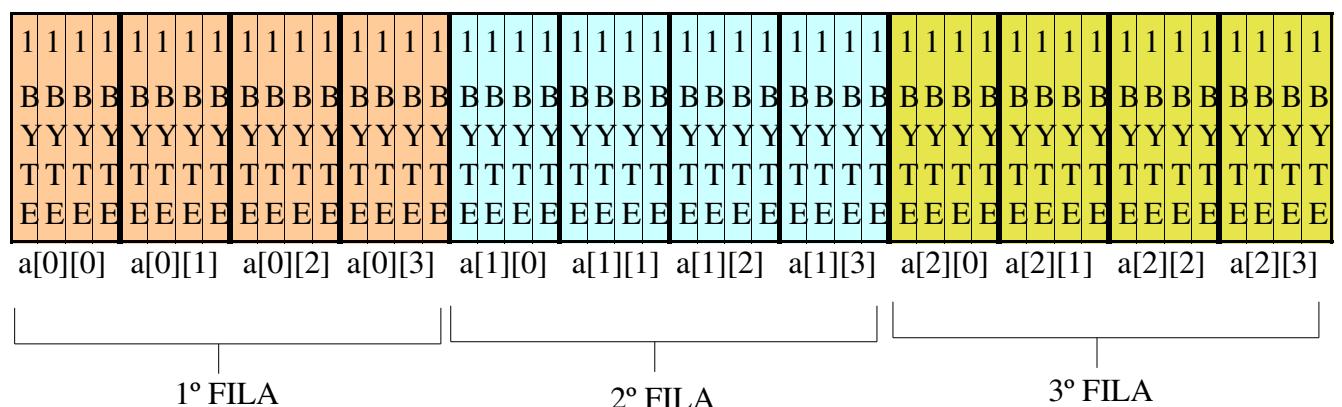
Una expresión de la forma

```
int a[3][4]           // Declara una variable a como un arreglo de 3 miembros, donde cada
                                              // miembro es a su vez un arreglo de 4 enteros
```

Los $3 \times 4 = 12$ elementos de este arreglo de 2 dimensiones son identificados mediante:

a[0][0], a[0][1], a[0][2], a[0][3], a[1][0], a[1][1], a[1][2], a[1][3], a[2][0], a[2][1], a[2][2], a[2][3]

Si imaginamos al arreglo de dos dimensiones como **una matriz A de orden 3x4**, es decir, **3 filas por 4 columnas**, entonces el **almacenamiento en memoria de los elementos se realiza “por filas”**. Es mejor visualizarlo con un esquema:

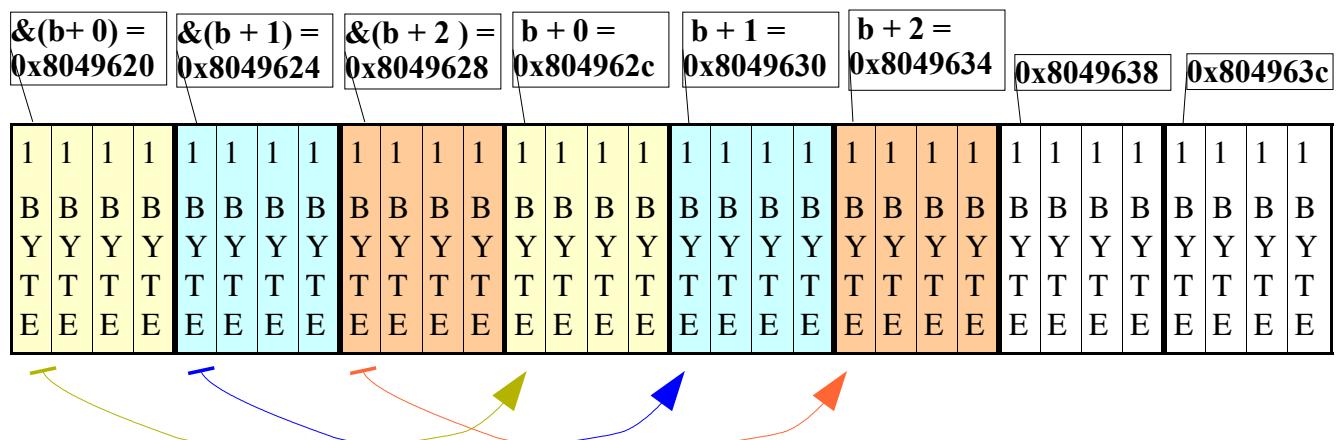


En cambio una expresión de la forma:

`int * b[3]; // Declara a la variable “b” cuyo tipo es “arreglo de 3 punteros a enteros”`

Cuyo respectivo esquema puede ser, dependiendo de como se valorizan los punteros:

De esta manera:

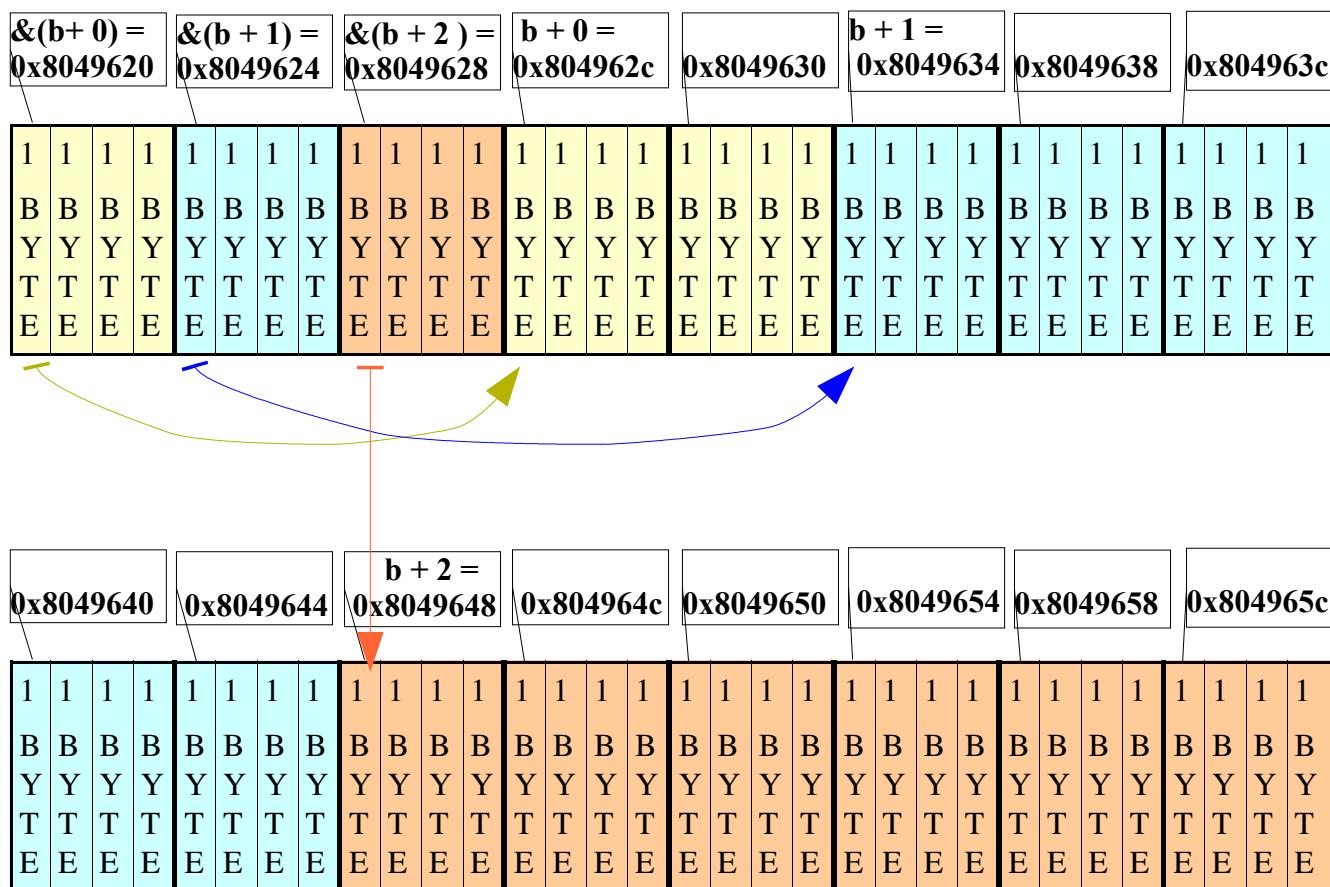


En este gráfico, observando las flechas que representan hacia dónde apuntan los punteros, podemos interpretar:

- $b = b + 0 = 0x804962c$ cuya dirección de memoria es $\&(b + 0) = 0x8049620$
- $b = b + 1 = 0x8049630$ cuya dirección de memoria es $\&(b + 1) = 0x8049624$
- $b = b + 2 = 0x8049634$ cuya dirección de memoria es $\&(b + 2) = 0x8049628$
- $*(b + 0)$ es un valor entero
- $*(b + 1)$ es un valor entero
- $*(b + 2)$ es un valor entero

Es decir que estamos ante un “**arreglo de 3 punteros a enteros**” cuando usamos la declaración: `int *b[3];` pero esto no estrictamente así, ya que una declaración como esta se puede también interpretar como se muestra en el siguiente esquema.

Otro posible esquema para la declaración int * b[3]; :



En este otro gráfico, observando las **flechas** que representan hacia dónde apuntan los punteros, podemos interpretar:

- $b = b + 0 = 0x804962c$ cuya dirección de memoria es $\&(b + 0) = 0x8049620$
- $b = b + 1 = 0x8049634$ cuya dirección de memoria es $\&(b + 1) = 0x8049624$
- $b = b + 2 = 0x8049648$ cuya dirección de memoria es $\&(b + 2) = 0x8049628$
- $*(b + 0)$ es un arreglo de 2 enteros
- $*(b + 1)$ es un arreglo de 5 enteros
- $*(b + 2)$ es un arreglo de 6 enteros

Es decir que estamos ante un “arreglo de 3 punteros a arreglos de enteros de distinto tamaño”

Por lo tanto **resulta evidente que no es lo mismo un “arreglo de arreglos” que un “arreglo de punteros”**, pues como se ve en los esquemas anteriores, **un “arreglo de arreglos” va a reservar espacio para la totalidad de sus elementos, mientras que “un arreglo de punteros” no actúa de esta manera debido a la diferente interpretación que se puede hacer con respecto a los elementos que están siendo apuntados.**

Veamos un **ejemplo** que muestra la **declaración y uso de “arreglo de arreglos”**:

```
// Archivo: arreglo-de-arreglos.c
#include <stdio.h>

#define FILAS 4
#define COLUMNAS 4

int main()
{
    int indice, fila, columna, a[]={1,2,3,4};
    int b[FILAS][COLUMNAS]={ {1,2,3,4},{5,6,7,8},{9,10,11,12},{13,14,15,16}};

    printf("\n");
    for( indice = 0 ; indice < FILAS ; indice++ )
    {
        printf("\n a[%d] = %d ", indice , a[indice] );
    }
    printf("\n");
    for( fila = 0 ; fila < FILAS ; fila++ )
    {
        printf("\n");
        for( columna = 0; columna < COLUMNAS ; columna++ )
        {
            printf(" b[%d][%d] = %d ", fila, columna, b[fila][columna] );
        }
        printf("\n");
    }
    return 0; // salida exitosa del programa
}

// Fin del archivo: arreglo-de-arreglos.c
```

Se compila con: **gcc -Wall arreglo-de-arreglos.c -o arreglo-de-arreglos.out**

Se ejecuta con: **./arreglo-de-arreglos.out**

Cuya salida en pantalla es:

```
a[0] = 1
a[1] = 2
a[2] = 3
a[3] = 4

b[0][0] = 1  b[0][1] = 2  b[0][2] = 3  b[0][3] = 4

b[1][0] = 5  b[1][1] = 6  b[1][2] = 7  b[1][3] = 8

b[2][0] = 9  b[2][1] = 10 b[2][2] = 11 b[2][3] = 12

b[3][0] = 13 b[3][1] = 14 b[3][2] = 15 b[3][3] = 16
```

Ahora veamos un sencillo programa que muestra el **uso de “arreglo de punteros a enteros”**:

```
// Archivo: arreglo-de-punteros.c
#include <stdio.h>
#include <stdlib.h>

#define SIZE 5
int main()
{
    int indice, * a[SIZE];
    int primera = 1, segunda = 2, tercera = 3, cuarta = 4, quinta = 5;
    a[0] = &primera;
    a[1] = &segunda;
    a[2] = &tercera;
    a[3] = &cuarta;
    a[4] = &quinta;
    printf("\n");
    for( indice = 0 ; indice < SIZE ; indice++ )
    {
        printf("\n a[ %d ] = %d en %p ", indice , *(a[indice]), a[indice] );
    }
    printf("\n");
    return 0; // salida exitosa del programa
}
//Fin del archivo: arreglo-de-punteros.c
```

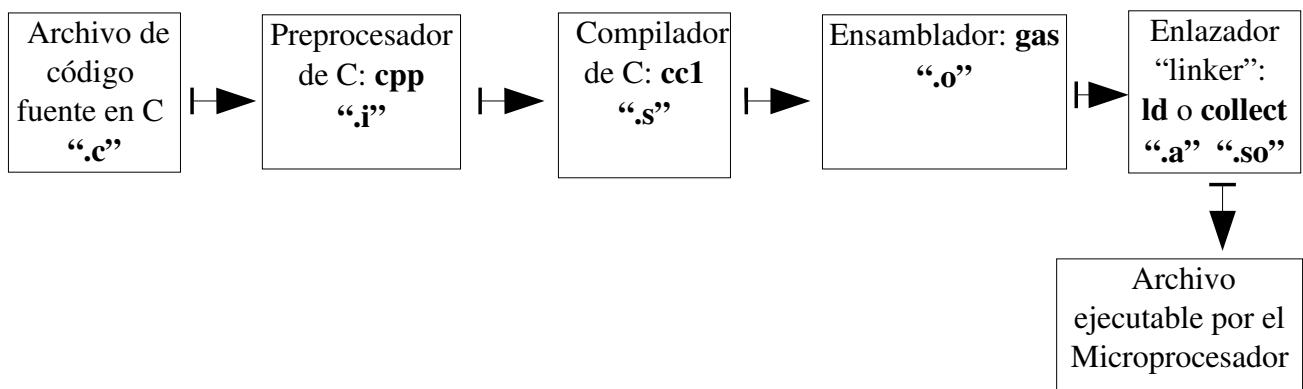
Se compila con: **gcc -Wall arreglo-de-punteros.c -o arreglo-de-punteros.out**

Se ejecuta con: **./arreglo-de-punteros.out**

Y su salida en pantalla es:

```
a[ 0 ] = 1 en 0xbffff74c
a[ 1 ] = 2 en 0xbffff748
a[ 2 ] = 3 en 0xbffff744
a[ 3 ] = 4 en 0xbffff740
a[ 4 ] = 5 en 0xbffff73c
```

Etapas del proceso de compilación:



Una manera de ver en pantalla cómo se va realizando el proceso de compilación es utilizar el argumento **-v** para la línea de comandos, cuando se está compilando con el **gcc**, cuyo efecto es activar el modo verborrágico del Compilador C de GNU.

Un ejemplo interesante es el siguiente:

```

// Archivo: qq.c
#include <stdio.h>

#define NULL ( (void *) 0 )

int main()
{
    char * p;
    p = malloc( 10 );
    if( p == NULL )
        return -1;
    return 0;
}
// Fin del archivo: qq.c
  
```

Cuando se compila de la siguiente manera: **gcc qq.c -Wall -E -v -o qq.i qq.c**

La opción **-E** detiene el proceso después de la etapa de preprocesamiento, y no realiza ninguna de las etapas posteriores, ni compilación, ni ensamblado, ni enlazado, y los archivos de salida contienen:

- **qq.i** : El código fuente en Lenguaje C que no necesita ser preprocesado
- **qq.c**: El mismo código fuente en Lenguaje C, el cuál sí necesita ser preprocesado

Cuando se compila de esta otra manera: **gcc qq.c -Wall -S -v -o qq.s**

La opción **-S** detiene el proceso después de la etapa de compilación, y no realiza ninguna de las etapas siguientes, ni ensamblado, ni enlazado, y el archivo de salida:

- **qq.s** : contiene el mismo código fuente pero en Lenguaje ensamblador

Entrada y Salida en Sistemas operativo derivados de UNIX:

A estos Sistemas Operativos se los suele llamar “**like *nix**”, es decir, **como UNIX**.

UNIX considera a muchas cosas como archivos.

La **Entrada y Salida está dividida en dos niveles: 1 y 2:**

- **printf** y **scanf** corresponden al **nivel 2**
- **open, read, write, y close, entre otros** corresponden al **nivel 1**

Veamos las del nivel 1:

¿Qué es abrir un archivo?

Es **disponer de su metadata**, la cuál está **compuesta de información extra que permite controlar los datos**. Esto se hace con:

```
int open( const char * nombre , int modo [ , int permisos ] )
```

Vamos a desglosar la información que hay en este **prototipo de la función open** haciendo uso de la información brindada por **man 2 open**:

Nota: El Nº 2 hace referencia a la sección dentro de la cuál se va a buscar el manual de open. Por lo general se presentan las siguientes secciones:

- Sección 1 Comandos de usuario
- Sección 2 Llamadas del sistema
- Sección 3 Subrutinas
- Sección 4 Dispositivos
- Sección 5 Formatos de archivos
- Sección 6 Juegos
- Sección 7 Varios
- Sección 8 Administración del sistema
- Sección 9 Núcleo

SINOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
int open(const char *camino, int flags);
int open(const char *camino, int flags, mode_t modo);
int creat(const char *camino, mode_t modo);
```

- **int** **open y creat devuelven el nuevo descriptor de fichero (file handler), o -1 si ha ocurrido un error** (en cuyo caso, se da un valor apropiado a **errno**). Observe que **open** puede abrir ficheros especiales de dispositivo, pero **creat** no puede crearlos; emplee **mknod(2)** en su lugar. En sistemas de ficheros NFS con asociación de UIDs habilitada, **open** puede devolver un descriptor de fichero pero p. ej. llamadas a **read(2)** pueden denegarse con el error **EACCES**. Esto es así porque el cliente hace el **open** comprobando los permisos, pero la asociación de UID la hace el servidor sobre las peticiones de lectura y escritura.
- **const char *camino** Es la **ruta absoluta, path, o camino donde se creará el fichero**, que quedará abierto, no compartido con ningún otro proceso

- **int flags** **flags** es uno de entre los siguientes modos de apertura del fichero:
 - **O_RDONLY** solamente para lectura
 - **O_WRONLY** solamente para escritura
 - **O_RDWR** para lectura y escritura
- **mode_t modo** Aquí se ha hecho uso del sinónimo: **typedef unsigned int mode_t;** El argumento **modo** especifica los permisos a emplear si se crea un nuevo fichero. Es modificado por la máscara **umask** del proceso de la forma habitual: los permisos del fichero creado son (modo & ~umask). Los **distintos modos de permisos** están **definidos como constantes simbólicas** en el **archivo fcntl.h**, estas son:
 - **S_IRWXU 00700** el usuario (el propietario del fichero) tiene permisos de lectura, escritura y ejecución
 - **S_IRUSR (S_IREAD)** **00400** el usuario tiene permiso de lectura
 - **S_IWUSR (S_IWRITE)** **00200** el usuario tiene permiso de escritura
 - **S_IXUSR (S_IEXEC)** **00100** el usuario tiene permiso de ejecución
 - **S_IRWXG 00070** el grupo tiene permiso de lectura, escritura y ejecución
 - **S_IRGRP** **00040** el grupo tiene permiso de lectura
 - **S_IWGRP** **00020** el grupo tiene permiso de escritura
 - **S_IXGRP** **00010** el grupo tiene permiso de ejecución
 - **S_IRWXO 00007** los otros tienen permiso de lectura, escritura y ejecución
 - **S_IROTH** **00004** los otros tienen permiso de lectura
 - **S_IWOTH** **00002** los otros tienen permiso de escritura
 - **S_IXOTH** **00001** los otros tienen permiso de ejecución

El argumento modo siempre debe especificarse cuando **O_CREAT** está en flags, y si no está, no es tenido en cuenta. **creat** equivale a **open** con **flags igual a O_CREAT | O_WRONLY | O_TRUNC.**

En **UNIX** los **archivos poseen un sistema de permisos** que sigue el siguiente esquema;

<u>r w x</u>	<u>r w x</u>	<u>r w x</u>	r: read, lectura o acceso
DUEÑO	GRUPO	OTROS	w: write: escritura o modificación
			x: ejecución

Otra función del nivel 1 es **read, la cuál lee de un descriptor de fichero:**

SINOPSIS

```
#include <unistd.h>
```

```
ssize_t read(int fd, void *buf, size_t nbytes);
```

Analicemos el prototipo de la función read:

- **ssize_t** Se ha hecho uso del **sinónimo: typedef unsigned int ssize_t;**
En caso de éxito, se devuelve el número de bytes leídos (cero indica fin de fichero), y el indicador de posición del fichero avanza este número de bytes. No es un error si este número es menor que el número de bytes pedidos; esto puede suceder por ejemplo porque ahora mismo haya disponible un número menor de bytes (quizás porque estamos cerca del fin-de-fichero, o porque estamos leyendo de una interconexión, o de una terminal), o porque read() ha sido interrumpido por una señal. **En caso de error, se devuelve -1, y se pone un valor apropiado en errno.** En este caso se deja indeterminado si el indicador de posición del fichero (si lo hay) cambia o no.
- **int fd** Es el **descriptor de fichero**
- **void *buf** Es un “**puntero a void**” (**puntero genérico**) que apunta al comienzo de una **zona de memoria** que se utiliza como bufer.
- **size_t nbytes** “**nbytes**” es el **número de bytes del archivo, cuyo descriptor de fichero es “fd”,** que se intenta leer y almacenar temporalmente en el bufer de lectura que se encuentra en **buf.** **Si nbytes es cero, read() devuelve cero** y no tiene otro efecto. **Si nbytes es mayor que SSIZE_MAX, el resultado es indefinido.**

Seguimos ahora con la **función, de nivel 1, write (ver man 2 write):**

NOMBRE

write - escribe a un descriptor de fichero

SINOPSIS

#include <unistd.h>

ssize_t write(int fd , const void *buf , size_t num);

DESCRIPCIÓN

write escribe hasta **num** bytes en el fichero referenciado por el descriptor de fichero **fd** desde el búfer que comienza en **buf.** **POSIX requiere que un read()** que pueda demostrarse que ocurra después que un **write()** haya regresado, devuelva los nuevos datos. Observe que no todos los sistemas de ficheros son conformes con **POSIX.**

El prototipo de la función write nos indica:

- **ssize_t** En caso de éxito, se devuelve el número de bytes escritos (cero indica pues que no se ha escrito nada). En caso de error, se devuelve -1 y se pone un valor apropiado en **errno**. Si num es cero y el descriptor de fichero se refiere a un fichero regular, se devolverá 0 sin que se cause ningún otro efecto. Para un fichero especial, los resultados no son transportables.
- **int fd** El fichero referenciado por el **descriptor de fichero fd**
- **const void *buf** Un “**puntero a void**” que apunta al comienzo del bufer de escritura
- **size_t num** Es el número de bytes que se quiere escribir en el fichero

Continuando con las funciones del **nivel 1**, seguimos con **close** (ver: man 2 close):

NOMBRE

close - cierra un descriptor de fichero

SINOPSIS

#include <unistd.h>

int close(int fd);

DESCRIPCIÓN escritura-lectura-ficheros.c

close cierra un descriptor de fichero de forma que ya no se refiera a fichero alguno y pueda ser reutilizado. Cualesquiera bloqueos mantenidos sobre el fichero con el que estaba asociado, y propiedad del proceso, son eliminados (sin importar qué descriptor de fichero fue utilizado para obtener el bloqueo).

Si fd es la última copia de cierto descriptor de fichero, los recursos asociados con dicho descriptor son liberados; si el descriptor fuera la última referencia a un fichero que haya sido eliminada mediante **unlink(2)** entonces el fichero es borrado.

EL prototipo de la función close nos indica:

- **int** **close** devuelve 0 en caso de éxito y -1 si ocurre algún error.
- **int fd** El descriptor del fichero que se quiere cerrar.

Otra de las funciones del **nivel 1** es **lseek** (ver: **man 2 lseek**):

NOMBRE

lseek - reposiciona el puntero de lectura/escritura de un fichero

SINOPSIS

```
#include <sys/types.h>
#include <unistd.h>
```

```
off_t lseek( int fildes , off_t offset , int whence );
```

DESCRIPCIÓN

La función **lseek** reposiciona el puntero del descriptor de fichero **fildes** con el argumento **offset** de acuerdo con la directiva **whence** tomando uno de los siguientes valores definidos como constantes simbólicas:

SEEK_SET El puntero se coloca a **offset bytes**.

SEEK_CUR El número de bytes indicado en **offset** se suma a la dirección actual y el puntero se coloca en la dirección resultante.

SEEK_END El puntero se coloca al final del fichero más **offset bytes**.

La función **lseek** permite colocar el puntero de fichero después del final de fichero. Si después se escriben datos en este punto, las lecturas siguientes de datos dentro del hueco que se forma devuelven ceros (hasta que realmente se escriban datos dentro de ese hueco).

El prototipo de la función **lseek** indica:

- **off_t** Se ha hecho uso del sinónimo: **typedef unsigned int off_t;**
En el caso de una ejecución correcta, **lseek** devuelve la posición del puntero resultante medida en bytes desde el principio del fichero. Si se produce un error, se devuelve el valor (**off_t**)-1 y en **errno** se coloca el tipo de error.
- **int fildes** El descriptor del fichero
- **off_t offset** Es el número de bytes en que se quiere desplazar el apuntador al descriptor del fichero
- **int whence** Es una directiva que puede tomar los valores definidos por las constantes simbólicas: “**SEEK_SET**”, “**SEEK_CUR**” y “**SEEK_END**”.

Ahora veamos un archivo de código fuente en Lenguaje C con ejemplos de lo que acabamos de ver:

// Archivo: apertura-escritura-lectura-cierre-ficheros.c

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

#define VECES 10
// Función: error      Muestra un mensaje de error en pantalla.
void error( char * s )
{
    perror( s );
    exit (-1); //salida con problemas
}
int main()
{
    double d;
    int i, fd, tmp;
    //long posicion;

    if( ( fd = open( "test.txt" , O_WRONLY|O_CREAT|O_TRUNC , 0666 ) ) < 0 )
    {
        error( "\nSe ha producido un error durante la llamada a open!!!\n");
        // Los números representados en octal (base 8) comienzan con CERO
    }
    for( i = 0 ; i < VECES ; i++ )
    {
        d = i + 0.5;
        if( write( fd , &d , sizeof(d) ) != sizeof(d) )
        {
            error( "\nSe ha producido un error durante la llamada a write!!!\n");
        }
    }
    close(fd); // Al cerrar el fichero se actualiza la metadata
    if( (fd = open( "test.txt" , O_RDONLY )) < 0 )
    {
        error( "\nSe ha producido un error durante la llamada a open!!!\n");
    }
    for( i = 0 ; i < VECES ; i++ )
    {
        if( (tmp = read( fd , &d , sizeof(d))) < 0 )
        {
            error( "\nSe ha producido un error durante la llamada a read!!!\n");
        }
        printf("\n %f " , d );
    }
    close(fd); // Al cerrar el fichero se actualiza la metadata
    return 0; // salida exitosa del programa
}
```

Se compila con:

```
gcc -Wall apertura-escritura-lectura-cierre-ficheros.c -o apertura-escritura-lectura-cierre-ficheros.out
```

Se ejecuta con: ./apertura-escritura-lectura-cierre-ficheros.out

Su salida en pantalla es:

```
0.500000  
1.500000  
2.500000  
3.500000  
4.500000  
5.500000  
6.500000  
7.500000  
8.500000  
9.500000
```

El archivo creado se llama: Nota: “**test.txt**”

Todo proceso recibe tres descriptores de ficheros que ya están abiertos:

- **0 Entrada estandar** “standard input” (típicamente el **teclado**)
- **1 Salida estandar** “standard output” (típicamente la **consola**)
- **2 Salida estandar** para mensajes **de error** (típicamente la **consola**)

Veamos un sencillo ejemplo:

// Archivo: entrada-salida-estandar.c

```
#include <unistd.h>

int main()
{
    char buffer[1024]; int cuanto;

    cuanto = read( 0 , buffer , sizeof(buffer) );
    // Lee desde el teclado y guarda en el descriptor de fichero 0
    // (entrada estandar: teclado
    // lo que hay en el buffer temporal de lectura

    buffer[cuento] = 0;
    buffer[0]++; // modifica el primer caracter del buffer

    write( 1 , buffer , cuanto );
    // Escribe lo que hay en el buffer en el archivo cuyo descriptor
    // de fichero es 1 (salida estandar: consola)

    return 0; // salida exitosa del programa
}
```

// Fin del archivo: entrada-salida-estandar.c

Se compila con: **gcc -Wall entrada-salida-estandar.c -o entrada-salida-estandar.out**
Se ejecuta con: **./entrada-salida-estandar.out**

Ahora veamos un ejemplo similar pero que hace **uso de la función lseek**:

Estamos suponiendo que queremos **leer los doubles que se almacenan en un archivo “test.txt” pero desde el último hacia el primero, o sea, al revés**.

// Archivo: lseek.c

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

#define VECES 10

// Función: error      Muestra un mensaje de error en pantalla.
void error( char * s )
{
    perror( s );
    exit (-1); //salida con problemas
}
int main()
{
    double d;
    int i, fd, retardo, tmp;
    long posicion;

    if( ( fd = open( "test.txt" , O_WRONLY|O_CREAT|O_TRUNC , 0666 ) ) < 0 )
    {
        error( "\nSe ha producido un error durante la llamada a open!!!\n");
        // Los números representados en octal (base 8) comienzan con CERO
    }
    for( i = 0 ; i < VECES ; i++ )
    {
        d = i + 0.5;
        if( write( fd , &d , sizeof(d) ) != sizeof(d) )
        {
            error( "\nSe ha producido un error durante la llamada a write!!!\n");
        }
    }
    close(fd); // Al cerrar el fichero se actualiza la metadata
    if( (fd = open( "test.txt" , O_RDONLY ))< 0 )
    {
        error( "\nSe ha producido un error durante la llamada a open!!!\n");
    }
```

```
// continúa el archivo: lseek.c
i=0;
posicion = lseek( fd , -sizeof(d) , SEEK_END );
do{
    read( fd , &d , sizeof(d) );
    printf("\n %f ", d );
    i++;
    lseek( fd , -2 * sizeof(d) , SEEK_CUR );
}while( i < VECES );

close(fd); // Al cerrar el fichero se actualiza la metadata
return 0; // salida exitosa del programa
}
// Fin del archivo: lseek.c
```

Se compila con: **gcc -Wall lseek.c -o lseek.out**

Se ejecuta con: **./lseek.out**

Y su salida en pantalla es:

```
9.500000
8.500000
7.500000
6.500000
5.500000
4.500000
3.500000
2.500000
1.500000
0.500000
```

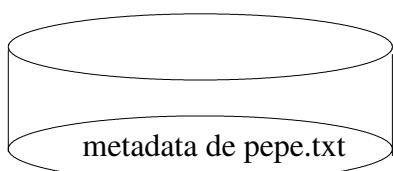
En los **sistemas *NIX** la memoria está distribuida respetando un esquema similar al siguiente:

ESPACIO DE USUARIO
open(“pepe.txt” , ... , ...)
// retorna “4”

ESPACIO DE KERNEL (NÚCLEO) DEL SO
Tabla de descriptores de ficheros:

0	ENTRADA ESTANDAR
1	SALIDA ESTANDAR
2	SALIDA ESTANDAR ERRORES
3	
4	pepe.txt

Y en el disco rígido se encuentra la metadata del archivo “pepe.txt” cuyo descriptor de fichero es “4”:



Pero: **¿Qué diferencia hay entre un programa y un proceso?**

Pues

un PROGRAMA es el archivo compilado ejecutable que está almacenado en el DISCO RÍGIDO;

mientras que

un PROCESO es este mismo archivo pero ejecutándose en MEMORIA.

En los sistemas *NIX se denota a un proceso de la siguiente manera:

PROCESO “P”



Dentro del SEGMENTO DE SISTEMA está la TABLA DE DESCRIPTORES DE FICHEROS.

Ahora vamos a ver una familia de funciones denominada exec (ver: man exec):

NOMBRE

execl, execlp, execle, execv, execvp - ejecutan un fichero

SINOPSIS

#include <unistd.h>

extern char **environ;

```
int execl( const char *camino, const char *arg, ...);
int execlp( const char *fichero, const char *arg, ...);
int execle( const char *camino, const char *arg , ..., char * const envp[]);
int execv( const char *camino, char *const argv[]);
int execvp( const char *fichero, char *const argv[]);
```

DESCRIPCIÓN

La familia de funciones exec reemplaza la imagen del proceso en curso con una nueva. Las funciones descritas en esta página del Manual son interfaces para la primitiva execve(2).

(Consulte la página del Manual de execve para información detallada acerca del reemplazo del proceso en curso.)

El primer argumento de estas funciones es el camino de un fichero que va a ser ejecutado. El const char *arg y puntos suspensivos siguientes en las funciones execl, execlp, y execle pueden ser contemplados como arg0, arg1, ..., argn. Todos juntos, describen una lista de uno o más punteros a cadenas de caracteres terminadas en cero, que representan la lista de argumentos disponible para el programa ejecutado. El primer argumento, por convenio, debe apuntar al nombre de fichero asociado con el fichero que se esté ejecutando. La lista de argumentos debe ser terminada por un puntero NULL.

Las funciones execv y execvp proporcionan un vector de punteros a cadenas de caracteres terminadas en cero, que representan la lista de argumentos disponible para el nuevo programa. El primer argumento, por convenio, debe apuntar al nombre de fichero asociado con el fichero que se esté ejecutando. El vector de punteros debe ser terminado por un puntero NULL.

La función execl también especifica el entorno del proceso ejecutado mediante un parámetro adicional que va detrás del puntero NULL que termina la lista de argumentos de la lista de parámetros o el puntero al vector argv. Este parámetro adicional es un vector de punteros a cadenas de caracteres acabadas en cero y debe ser terminada por un puntero NULL. Las otras funciones obtienen el entorno para la nueva imagen de proceso de la variable externa environ en el proceso en curso.

Algunas de estas funciones tienen una semántica especial.

Las funciones execlp y execvp duplicarán las acciones del shell al buscar un fichero ejecutable si el nombre de fichero especificado no contiene un carácter de barra inclinada (/). El camino de búsqueda es el especificado en el entorno por la variable PATH. Si esta variable no es especificada, se emplea el camino predeterminado ``:/bin:/usr/bin''. Además, ciertos errores se tratan de forma especial.

Si a un fichero se le deniega el permiso (la función intentada execve devuelve EACCES), estas funciones continuarán buscando en el resto del camino de búsqueda. Si no se encuentra otro fichero, empero, retornarán dando el valor EACCES a la variable global errno.

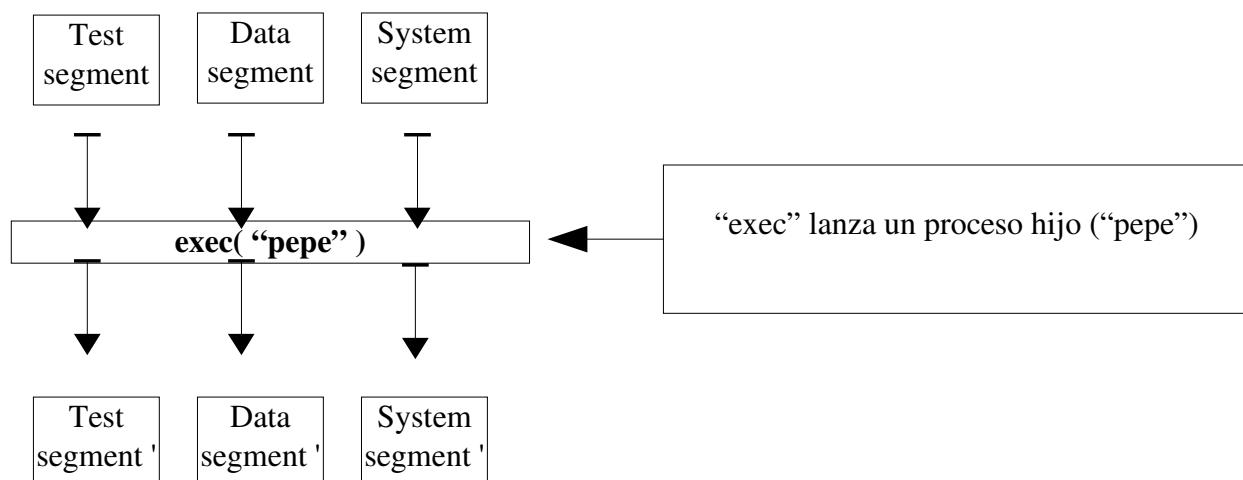
Si no se reconoce la cabecera de un fichero (la función intentada execve devuelve ENOEXEC), estas funciones ejecutarán el shell con el camino del fichero como su primer argumento. (Si este intento falla, no se busca más.)

VALOR DEVUELTO

Si cualquiera de las funciones exec regresa, es que ha ocurrido un error. El valor de retorno es -1, y en la variable global errno se pondrá el código de error adecuado.

¿En qué puede afectar el esquema de exec?

PROCESO P



En el segmento de sistema está la tabla de descriptores de ficheros. El proceso P' "hereda" la tabla de descriptores de ficheros del proceso anterior TAL COMO ESTÁ

La primitiva dup (ver: man 2 dup):

DUP(2)

Manual del Programador de Linux

DUP(2)

NOMBRE

dup, dup2 - duplica un descriptor de fichero

SINOPSIS

#include <unistd.h>

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

DESCRIPCIÓN

dup y dup2 crean una copia del descriptor de fichero oldfd.

Después de una llamada a dup o dup2 con éxito, los descriptores antiguo y nuevo pueden usarse indiferentemente.

Comparten candados (locks), indicadores de posición de fichero y banderas (flags); por ejemplo, si la posición del fichero se modifica usando lseek en uno de los descriptores, la posición en el otro también cambia.

Sin embargo los descriptores no comparten la bandera close-on-exec, (cerrar-al-ejecutar).

dup usa el descriptor libre con menor numeración posible como nuevo descriptor.

dup2 hace que newfd sea la copia de oldfd, cerrando primero newfd si es necesario.

VALOR DEVUELTO

dup y dup2 devuelven el valor del nuevo descriptor, ó -1 si ocurre algún error, en cuyo caso errno toma un valor apropiado.

ERRORES

EBADF oldfd no es un descriptor de fichero abierto, o newfd está fuera del rango permitido para descriptores de ficheros.

EMFILE El proceso ya tiene el máximo número de descriptores de fichero abiertos y se ha intentado abrir uno

ADVERTENCIA

El error devuelto por dup2 es diferente del devuelto por fcntl(..., F_DUPFD,...) cuando newfd está fuera de rango. En algunos sistemas dup2 a veces devuelve EINVAL como F_DUPFD.

CONFORME A

SVID, AT&T, POSIX, X/OPEN, BSD 4.3. SVr4 documenta las condiciones de error adicionales EINTR y ENOLINK. **POSIX.1** añade EINTR.

VÉASE TAMBIÉN

fcntl(2), open(2), close(2)

La primitiva **dup** duplica el descriptor correspondiente a fd. Garantiza que el duplicado quedará en la primera entrada libre (contando desde cero).

Veamos un ejemplo en el siguiente código fuente:

```
// Archivo: dup-redireccion-entrada-salida.c

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main()
{
    int fd = open( "test2.txt" , O_WRONLY|O_CREAT|O_TRUNC , 0666 );
    /* Tabla de descriptores de ficheros:
     * 0   STDIN
     * 1   STDOUT
     * 2   STDERR
     * .
     * .
     * fd   test2.txt
     * .
     */
    close( 1 ); // cierra STDOUT
    /* Tabla de descriptores de ficheros:
     * 0   STDIN
     * 1       {CERRADO queda libre
     * 2   STDERR
     * .
     * .
     * fd   test2.txt
     * .
     */
    dup( fd );
    /* Tabla de descriptores de ficheros:
     * 0   STDIN
     * 1   test2.txt  {dup le da el menor
     * 2   STDERR
     * .
     * .
     * fd   test2.txt
     * .
     */
}
```

```
// continúa el archivo: dup-redireccion-entrada-salida.c
close( fd );
/* Tabla de descriptores de ficheros:
 * 0    STDIN
 * 1    test2.txt  {dup le da el menor
 * 2    STDERR
 *
 */
* fd      {CERRADO queda libre
*
*/
}

execl( "/bin/ls" , "/bin/ls" , NULL );
perror( " ERROR " );
return 0; // salida satisfactoria del programa
}
// Fin del archivo: dup-redireccion-entrada-salida.c
```

Se compila con: **gcc -Wall dup-redireccion-entrada-salida.c -o dup-redireccion-entrada-salida.out**

Se ejecuta con: **./dup-redireccion-entrada-salida.out**

Este programa hace una **REDIRECCIÓN DE ENTRADA – SALIDA**, lo que con el shell bash se haría mediante el comando:

ls > test2.txt

Este es el **mecanismo para redirigir la Entrada y la Salida usado por el shell respetando la siguiente manera:**

programa > archivo
programa < archivo
programa >> archivo
programa << archivo

Las dos últimas formas corresponden al formato “APPEND”, es decir, SE LE AGREGA A PARTIR DE LA PARTE FINAL DEL ARCHIVO.

Ahora veamos los PIPEs. Los PIPEs son un ejemplo de algo llamado **INTERPROCESS COMUNICATION**.

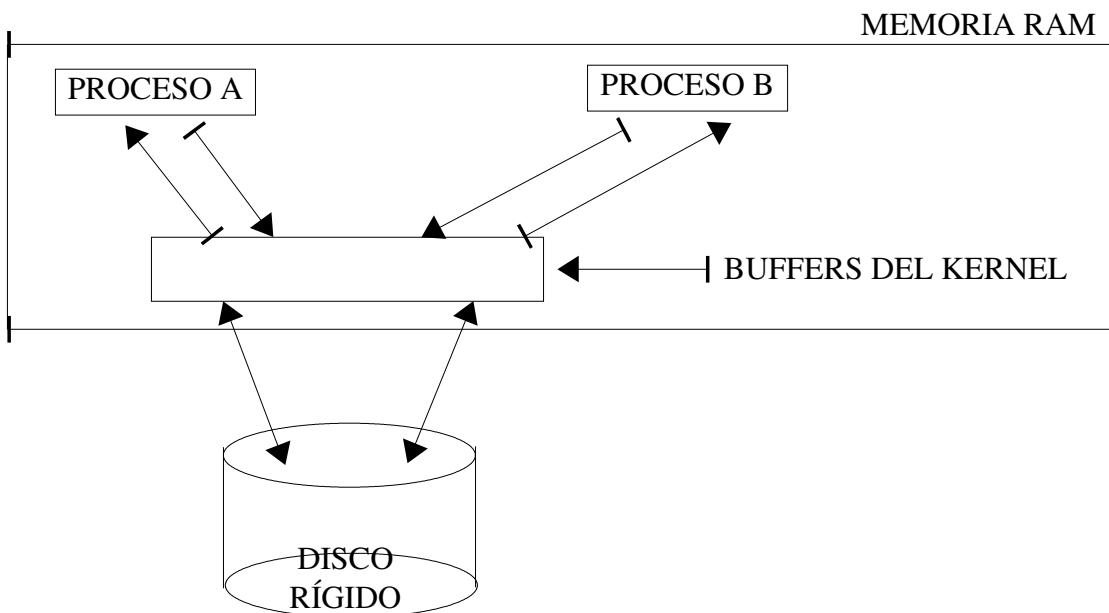
¿Cómo pueden comunicarse (pasarse datos) dos procesos?

Una forma es mediante el control de una persona.

Otra forma es mediante un archivo en disco rígido.

Pero aquí hay que notar que lo que el “**write**” nos asegura es que se van a escribir los bytes en un buffer del sistema operativo y no directamente sobre el propio disco rígido, el “**write**” y el “**open**” deben su eficiencia a esta característica. Después de algún tiempo los bytes se van a escribir al disco rígido.

Entonces podemos tratar de “cortocircuitar” la comunicación prescindiendo del archivo real en disco rígido.



Ventajas:

- Los procesos “creen” estar trabajando con archivos del disco rígido, (podemos usar las funciones “read” y “write”)

Este es el mecanismo que se conoce como PIPE, tubería en español.

Veamos que nos informa la página del manual (ver: **man 2 pipe**):

PIPE(2)

Manual del Programador de Linux

PIPE(2)

NOMBRE

pipe - crea una tubería o interconexión

SINOPSIS

```
#include <unistd.h>
```

```
int pipe(int descf[2]);
```

DESCRIPCIÓN

pipe crea un par de descriptores de ficheros, que apuntan a un nodo-í de una tubería, y los pone en el vector de dos elementos apuntado por descf. descf[0] es para lectura, descf[1] es para escritura.

VALOR DEVUELTO

En caso de éxito, se devuelve cero. En caso de error se devuelve -1 y se pone un valor apropiado en errno.

ERRORES

EMFILE El proceso tiene en uso demasiados descriptores de ficheros.

ENFILE La tabla de ficheros del sistema está llena.

EFAULT descf no es válido.

CONFORME A

SVr4, SVID, AT&T, POSIX, X/OPEN, BSD 4.3

VÉASE TAMBIÉN

read(2), write(2), fork(2), socketpair(2)

Un pipe se crea con: **int pipe(int fd[2]);**

Retorna 0 si la creación del pipe, es decir, los dos descriptores de ficheros se realizó satisfactoriamente:

- **fd[0] es solo para lectura**
- **fd[1] es solo para escritura**

Retorna -1 si se produjo algún error.

El Sistema Operativo aloja un buffer para cada pipe.

Veamos un ejemplo:

```
// Archivo: pipe.c
#include <stdio.h>
#include <unistd.h>

#define SIZE 1024

int main()
{
    int pip[ 2 ], cuanto;
    char linea[SIZE];
    pipe( pip );
    write( pip[1] , "HOLA MUNDO!!!" , 13 );
    cuanto = read( pip[0] , linea , SIZE );
    linea[cuanto] = 0;
    printf("[%"s"\n" , linea );
    close( pip[0] );
    close( pip[1] );
    return 0; // salida exitosa del programa
}
// Fin del archivo: pipe.c
```

Que se compila con: **gcc -Wall pipe.c -o pipe.out**

Se ejecuta con: **./pipe.out**

Su salida en pantalla es: **[HOLA MUNDO!!!]**

Este programa se pasa “HOLA MUNDO!!!” a sí mismo.

Ahora veamos otra primitiva denominada “**fork**” (ver: **man 2 fork**)

FORK(2)
NOMBRE

Manual del Programador de Linux

FORK(2)

fork - crean un proceso hijo
SINOPSIS

```
#include <unistd.h>
```

```
pid_t fork(void);
```

DESCRIPCIÓN

fork crea un proceso hijo que difiere de su proceso padre sólo en su PID y PPID, y en el hecho de que el uso de recursos esté asignado a 0. Los candados de fichero (file locks) y las señales pendientes no se heredan.

En linux, fork está implementado usando páginas de copia-en-escritura (copy-on-write), así que la única penalización en que incurre fork es en el tiempo y memoria requeridos para duplicar las tablas de páginas del padre, y para crear una única estructura de tarea (task structure) para el hijo.

VALOR DEVUELTO

En caso de éxito, se devuelve el PID del proceso hijo en el hilo de ejecución de su padre, y se devuelve un 0 en el hilo de ejecución del hijo. En caso de fallo, se devolverá un -1 en el contexto del padre, no se creará ningún proceso hijo, y se pondrá en errno un valor apropiado.

ERRORES

EAGAIN fork no puede reservar suficiente memoria para copiar las tablas de páginas del padre y alojar una estructura de tarea para el hijo.

ENOMEM fork no pudo obtener las necesarias estructuras del núcleo porque la cantidad de memoria era escasa.

CONFORME A

La llamada al sistema fork es conforme con SVr4, SVID, POSIX, X/OPEN y BSD 4.3.

VÉASE TAMBIÉN

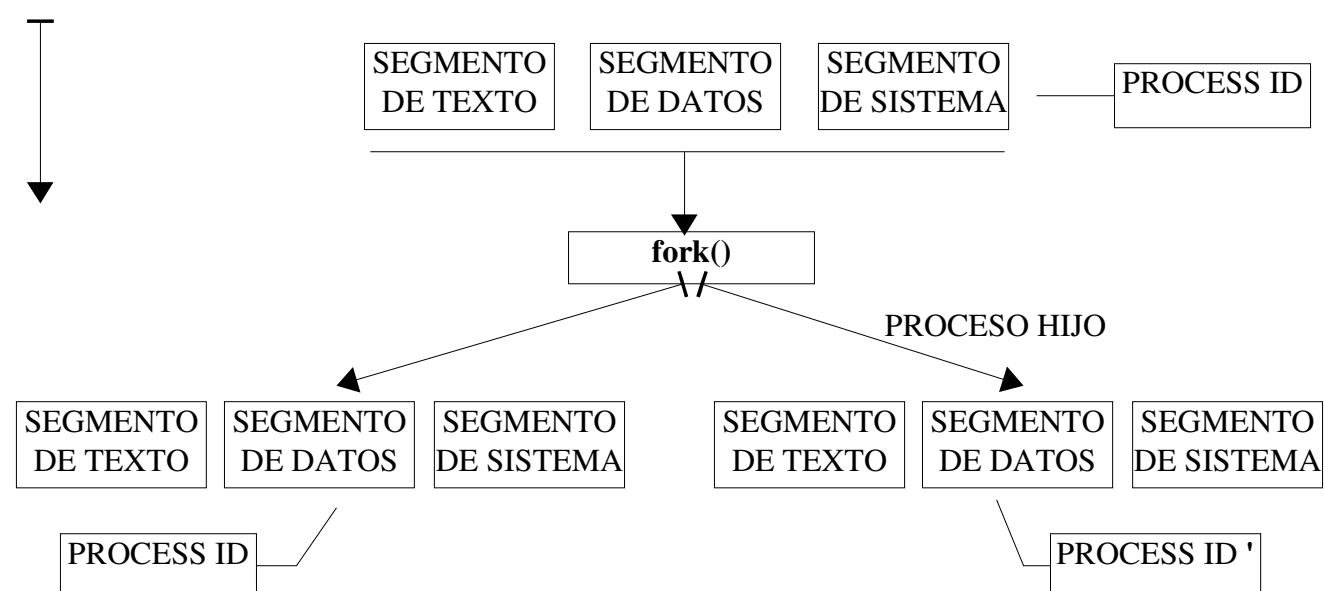
clone(2), execve(2), wait(2)

Linux 1.2.9

1 julio 1996

FORK(2)

TIEMPO



Un ejemplo de uso de la primitiva “**fork**”:

```
// Archivo: fork.c
#include <stdio.h>
#include <unistd.h>

int main()
{
    printf("Vamos... \n");
    if( fork() != 0 )
        printf("PADRE!!!\n");
    else
        printf("HIJO!!!\n");
    printf("Salimos...\n");
    return 0;
}
// Fin del archivo: fork.c
```

Que se compila con: **gcc -Wall fork.c -o fork.out**

Se ejecuta con: **./fork.out**

Y su salida en pantalla es:

```
Vamos...
HIJO!!!
Salimos...
PADRE!!!
Salimos...
```

Primitivas “exit” y “wait”:

Veamos la página de manual de la primitiva “wait” (**man 2 wait**):

WAIT(2) **Manual del Programador de Linux**

WAIT(2)

NOMBRE

wait, waitpid - espera por el final de un proceso

SINOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status, int options);
```

DESCRIPCIÓN

La función **wait** suspende la ejecución del proceso actual hasta que un proceso hijo ha terminado, o hasta que se produce una señal cuya acción es terminar el proceso actual o llamar a la función manejadora de la señal. Si un hijo ha salido cuando se produce la llamada (lo que se entiende por proceso "zombie"), la función vuelve inmediatamente. Todos los recursos del sistema reservados por el hijo son liberados.

La función waitpid suspende la ejecución del proceso en curso hasta que un hijo especificado por el argumento pid ha terminado, o hasta que se produce una señal cuya acción es finalizar el proceso actual o llamar a la función manejadora de la señal.

Si el hijo especificado por pid ha terminado cuando se produce la llamada (un proceso "zombie"), la función vuelve inmediatamente. Todos los recursos del sistema reservados por el hijo son liberados.

El valor de pid puede ser uno de los siguientes:

< -1 lo que significa esperar a que cualquier proceso hijo cuyo ID del proceso es igual al valor absoluto de pid.

-1 lo que significa que espera por cualquier proceso hijo; este es el mismo comportamiento que tiene wait.

0 lo que significa que espera por cualquier proceso hijo cuyo ID es igual al del proceso llamante.

> 0 lo que significa que espera por el proceso hijo cuyo ID es igual al valor de pid.

El valor de options es un OR de cero o más de las siguientes constantes:

WNOHANG

que significa que vuelve inmediatamente si ningún hijo ha terminado.

WUNTRACED

que significa que también vuelve si hay hijos parados, y de cuyo estado no ha recibido notificación.

Si status no es NULL, wait o waitpid almacena la información de estado en la memoria apuntada por status.

Si el estado puede ser evaluado con las siguientes macros (dichas macros toman el buffer stat (un int) como argumento -- ¡no un puntero al buffer!):

WIFEXITED(status)

es distinto de cero si el hijo terminó normalmente.

WEXITSTATUS(status)

evalúa los ocho bits menos significativos del código de retorno del hijo que terminó, que podrían estar activados como el argumento de una llamada a exit() o como el argumento de un return en el programa principal. Esta macro solamente puede ser tenida en cuenta si WIFEXITED devuelve un valor distinto de cero.

WIFSIGNALED(status)

devuelve true si el proceso hijo terminó a causa de una señal no capturada.

WTERMSIG(status)

devuelve el número de la señal que provocó la muerte del proceso hijo. Esta macro sólo puede ser evaluada si WIFSIGNALED devolvió un valor distinto de cero.

WIFSTOPPED(status)

devuelve true si el proceso hijo que provocó el retorno está actualmente pardo; esto solamente es posible si la llamada se hizo usando WUNTRACED.

WSTOPSIG(status)

devuelve el número de la señal que provocó la parada del hijo. Esta macro solamente puede ser evaluada si WIFSTOPPED devolvió un valor distinto de cero.

VALOR DEVUELTO

El ID del proceso del hijo que terminó, -1 en caso de error o cero si se utilizó WNOHANG y no hay hijo disponible (en este caso, errno se pone a un valor apropiado).

ERRORES

ECHILD si el proceso especificado en pid no termina o no es hijo del proceso llamante. (Esto puede ocurrir para nuestros propios hijos si se asigna SIG_IGN como acción de SIGCHLD.)

EINVAL si el argumento options no fue valido.

ERESTARTSYS

si no se activó WNOHANG y si no se ha capturado una señal no bloqueada o SIGCHLD. El interfaz de la biblioteca no tiene permitido devolver ERESTARTSYS, pero devolverá EINTR.

NOTAS

The Single Unix Specification (Especificación para un Unix Único) describe un modificador SA_NOCLDWAIT (no presente en Linux) tal que si este modificador está activo, o bien se ha asignado SIG_IGN como acción para SIGCHLD (que, por cierto, no está permitido por POSIX), entonces los hijos que terminan no se convierten en zombies y una llamada a wait() o waitpid() se bloqueará hasta que todos los hijos hayan terminado y, a continuación, fallará asignando a errno el valor ECHILD.

CONFORME A

SVr4, POSIX.1

VÉASE TAMBIÉN

signal(2), wait4(2), signal(7)

Linux

23 junio 1997

WAIT(2)

La función “wait” bloquea el proceso hasta que un (1) hijo termina. Si el proceso no tiene hijos wait devuelve un valor distinto de CERO indicando un error. Su “status” (que es el argumento de “exit”) queda en “status” y devuelve el PID del hijo.

Veamos la página de manual de la primitiva “**exit**” :

man 3 exit nos proporciona la siguiente información:

EXIT(3)

Manual del Programador de Linux

EXIT(3)

NOMBRE

exit - produce la terminación normal del programa

SINOPSIS

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

```
void exit(int status);
```

DESCRIPCIÓN

La función **exit** produce la terminación normal del programa y la devolución de **status** al proceso padre. Antes, se llama a todas las funciones registradas con **atexit()** y **on_exit()** en orden inverso a su registro, y todos los flujos abiertos se vuelcan y cierran.

VALOR DEVUELTO

La función **exit()** no devuelve nada ni regresa.

CONFORME A

SVID 3, POSIX, BSD 4.3, ISO 9899

VÉASE TAMBIÉN

_exit(2), atexit(3), on_exit(3)

GNU

9 de Enero de 1998

EXIT(3)

man 2 exit nos muestra:

_EXIT(2)

Manual del programador de Linux

_EXIT(2)

NOMBRE

_exit - Produce la terminación del proceso actual

SINOPSIS

```
#include <unistd.h>
```

```
void _exit(int status);
```

DESCRIPCIÓN

_exit termina inmediatamente la ejecución del proceso invocador. Todos los descriptores de ficheros abiertos que pertenezcan al proceso se cierran; todos los procesos hijos son heredados por el proceso 1, init, y al proceso padre se le envía la señal **SIGCHLD**.

status es el valor que se le devuelve al proceso padre como estado de terminación del proceso, y se puede leer mediante una de las funciones de la familia de **wait**.

VALOR DEVUELTO

_exit nunca regresa.

CONFORME A

SVID, AT&T, POSIX, X/OPEN, BSD 4.3

NOTAS

_exit no llama a ninguna función registrada mediante la función estándar atexit del ANSI C y no vacía los almacenamientos temporales (buffers) de E/S estándares. Para hacer esto utilice exit(3).

VÉASE TAMBIÉN

fork(2), execve(2), waitpid(2), wait4(2), kill(2), wait(2), exit(3)

Linux

21 Julio 1993

_EXIT(2)

La función “exit” termina el proceso limpiamente, es decir, cierra todos los archivos, libera todos los recursos, etc).

ANSI C establece que : return n ≡ exit(n)

Veamos un ejemplo:

// Archivo: exit-wait.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    int pid;
    if( ( pid = fork() ) == 0 ) /* hijo */
    {
        sleep( 15 );
        return 15; // equivale a exit(15)
    }
    else /* padre */
    {
        int status, p ;
        printf("ESPERAMOS a %d \n" , pid );
        p = wait( &status );
        printf("de %d llega %d \n" , p , status );
    }
    return 0; // salida exitosa del programa
}

// Fin del archivo: exit-wait.c
```

Que se compila con: gcc -Wall exit-wait.c -o exit-wait.out

Y Se ejecuta con:

./exit-wait.out

Cuya salida en pantalla es:

**ESPERAMOS a 4581
de 4581 llega 3840**

Ahora si le hacemos una leve modificación:

// Archivo: exit-wait-2.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    int pid;
    if( ( pid = fork() ) == 0 ) /* hijo */
    {
        sleep( 15 );
        return 15; // equivale a exit(15)
    }
    else /* padre */
    {
        int status, p ;
        printf("ESPERAMOS a %d \n" , pid );
        p = wait( &status );
        printf("de %d llega %d \n" , p , WEXITSTATUS( status ) );
    }
    return 0; // salida exitosa del programa
}
// Fin del archivo: exit-wait-2.c
```

Que se compila con:

gcc -Wall exit-wait-2.c -o exit-wait-2.out

Y Se ejecuta con:

./exit-wait-2.out

Su salida en pantalla es ahora:

**ESPERAMOS a 4643
de 4643 llega 15**

Ahora usemos “**pipe**” con “**fork**”:

// Archivo: fork-pipe.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    int pip[2];
    pipe( pip );
    if( fork() ) /* padre */
    {
        write( pip[1] , "HOLA MUNDO\n" , 12 );
        wait( NULL );
    }
    else /* hijo */
    {
        char l[1024];
        int cuanto = read( pip[0] , 1 , sizeof( 1 ) );
        l[cuanto] = 0; // indica fin
        printf("llega [ %s ] \n" , l );
    }
    return 0; // salida exitosa del programa
}
// Fin del archivo: fork-pipe.c
```

Que se compila con:

gcc -Wall fork-pipe.c -o fork-pipe.out

Se ejecuta con:

./fork-pipe.out

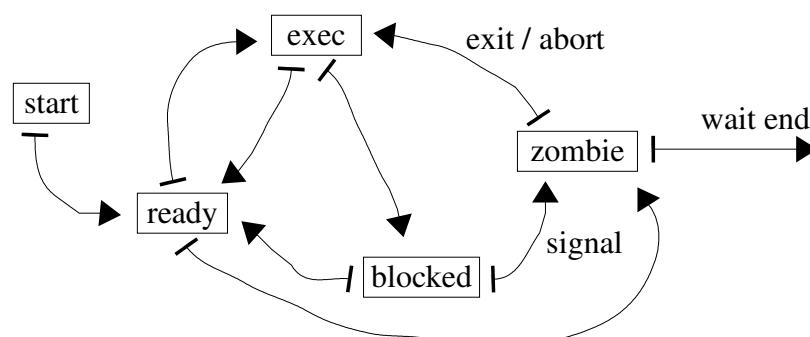
Y su salida en pantalla es:

llega [HOLA MUNDO

]

Ahora veamos: **¿en qué estado puede estar un proceso?**

- **Ejecutándose (exec)**
- **no ejecutándose pero “ready”, listo para ser ejecutado**



Señales:

Tienen que ver con eventos que son más o menos inesperados. **Eventos asíncronos.**

Estos eventos pueden ser errores, condiciones que sin llegar a ser errores son inesperadas que aparezcan. Excepciones de otras cosas (**exit** de un hijo, presencia de datos para leer, etc).

UNIX, tradicionalmente los ha llamado SIGNALS (señales) a la aparición de estos eventos; los ha caracterizado en 32 clases (actualmente hay 64 por inclusión de señales de tiempo real). Cada señal produce un efecto por default en el proceso que las recibe típicamente, o son ignoradas, o abortan el proceso. UNIX permite tomar otra acción frente a una señal: se puede asociar a una señal una función (signal handler) que se ejecuta cuando la señal aparece.

Esto se logra con:

```
#include <signal.h>
```

La página del **man 2 signal** nos proporciona la siguiente información:

SIGNAL(2)

Manual del Programador de Linux

SIGNAL(2)

NOMBRE

signal - manejo de señales según C ANSI

SINOPSIS

```
#include <signal.h>
```

```
void (*signal(int signum, void (*manejador)(int)))(int);
```

DESCRIPCIÓN

La llamada al sistema **signal** instala un nuevo manejador de señal para la señal cuyo número es **signum**. El manejador de señal se establece como manejador, que puede ser una función especificada por el usuario, o una de las siguientes macros:

SIG_IGN

No tener en cuenta la señal.

SIG_DFL

Dejar la señal con su comportamiento predefinido.

El argumento entero que se pasa a la rutina de manejo de señal es el número de la señal. Esto hace posible emplear un mismo manejador de señal para varias de ellas.

Los manejadores de señales son rutinas que se llaman en cualquier momento en el que el proceso recibe la señal correspondiente. Usando la función **alarm(2)**, que envía una señal **SIGALRM** al proceso, es posible manejar fácilmente trabajos regulares. A un proceso también se le puede decir que relea sus ficheros de configuración usando un manejador de señal (normalmente, la señal es **SIGHUP**).

VALOR DEVUELTO

signal devuelve el valor anterior del manejador de señal, o **SIG_ERR** si ocurre un error.

OBSERVACIONES

No se pueden instalar manejadores para las señales **SIGKILL** ni **SIGSTOP**.

Desde libc6, signal usa la semántica BSD y el comportamiento por defecto es no reasignar una señal a su comportamiento por defecto. Puede usar sysv_signal para obtener la semántica SysV.

Ambas, signal and sysv_signal son rutinas de biblioteca construidas sobre sigaction(2).

Si usted no entiende bien el prototipo del principio de esta página, puede ayudar el verlo separado así:

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t manejador);
```

De acuerdo con POSIX, el comportamiento de un proceso es indefinido después de no hacer caso a una señal SIGFPE, SIGILL o SIGSEGV que no ha sido generada por las funciones kill(2) ni raise(2). La división entera por cero tiene un resultado indefinido. En algunas arquitecturas generará una señal SIGFPE. (También, el dividir el entero más negativo por -1 puede generar SIGFPE.) No hacer caso a esta señal puede conducir a un bucle infinito.

De acuerdo con POSIX (B.3.3.1.3) no debe asignar SIG_IGN como acción para SIGCHLD. Aquí los comportamiento de BSD y SYSV difieren, provocando el fallo en Linux de aquellos programas BSD que asignan SIG_IGN como acción para SIGCHLD.

CONFORME A C ANSI

VÉASE TAMBIÉN

kill(1), kill(2), killpg(2), pause(2), raise(3), sigaction(2), signal(7), sigsetops(3), sigvec(2), alarm(2)

Linux 2.0

21 agosto 1997

SIGNAL(2)

Y esta otra página del manual **man 7 signal** nos indica:

SIGNAL(7)

Manual del Programador de Linux

SIGNAL(7)

NOMBRE

signal - lista de las señales disponibles

DESCRIPCIÓN

Linux permite el uso de las señales dadas a continuación. Los números de varias de las señales dependen de la arquitectura del sistema. Primero, las señales descritas en POSIX.1.

Señal	Valor	Acción	Comentario
<hr/>			
SIGHUP	1	A	Cuelgue detectado en la terminal de control o muerte del proceso de control
SIGINT	2	A	Interrupción procedente del teclado
SIGQUIT	3	C	Terminación procedente del teclado
SIGILL	4	C	Instrucción ilegal
SIGABRT	6	C	Señal de aborto procedente de abort(3)

SIGFPE	8	C	Excepción de coma flotante
SIGKILL	9	AEF	Señal de matar
SIGSEGV	11	C	Referencia inválida a memoria
SIGPIPE	13	A	Tubería rota: escritura sin lectores
SIGALRM	14	A	Señal de alarma de alarm(2)
SIGTERM	15	A	Señal de terminación
SIGUSR1	30,10,16	A	Señal definida por usuario 1
SIGUSR2	31,12,17	A	Señal definida por usuario 2
SIGCHLD	20,17,18	B	Proceso hijo terminado o parado
SIGCONT	19,18,25		Continuar si estaba parado
SIGSTOP	17,19,23	DEF	Parar proceso
SIGTSTP	18,20,24	D	Parada escrita en la tty
SIGTTIN	21,21,26	D	E. de la tty para un proc. de fondo
SIGTTOU	22,22,27	D	S. a la tty para un proc. de fondo

A continuación las señales que no están en POSIX.1 pero descritas en SUSv2.

Señal	Valor	Acción	Comentario
SIGBUS	10,7,10	C	Error de bus (acceso a memoria inválido)
SIGPOLL		A	Evento que se puede consultar (Sys V).
			Sinónimo de SIGIO
SIGPROF	27,27,29	A	Ha expirado el reloj de perfilado (profiling)
SIGSYS	12,-,12	C	Argumento de rutina inválido (SVID)
SIGTRAP	5	C	Trampa de traza/punto de ruptura
SIGURG	16,23,21	B	Condición urgente en conector (4.2 BSD)
SIGVTALRM	26,26,28	A	Alarma virtual (4.2 BSD)
SIGXCPU	24,24,30	C	Límite de tiempo de CPU excedido (4.2 BSD)
SIGXFSZ	25,25,31	C	Límite de tamaño de fichero excedido (4.2 BSD)

(Para los casos SIGSYS, SIGXCPU, SIGXFSZ y, en algunas arquitecturas, también SIGBUS, la acción por omisión en Linux hasta ahora (2.3.27) es A (terminar), mientras que SUSv2 prescribe C (terminar y volcado de memoria)).

A continuación otras señales.

Señal	Valor	Acción	Comentario
SIGIOT	6	C	Trampa IOT. Un sinónimo de SIGABRT
SIGEMT	7,-,7		
SIGSTKFLT	-,16,-	A	Fallo de la pila en el coprocesador
SIGIO	23,29,22	A	E/S permitida ya (4.2 BSD)
SIGCLD	-,18		Un sinónimo de SIGCHLD
SIGPWR	29,30,19	A	Fallo de corriente eléctrica (System V)
SIGINFO	29,-,-		Un sinónimo para SIGPWR
SIGLOST	-,,-	A	Bloqueo de fichero perdido.

SIGWINCH	28,28,20	B	Señal de reescalado de la ventana (4.3 BSD, Sun)
SIGUNUSED	-,31,-	A	Señal no usada.

(Aquí, - denota que una señal está ausente. Allí donde se indican tres valores, el primero es comúnmente válido para alpha y sparc, el segundo para i386, ppc y sh, y el último para mips. La señal 29 es SIGINFO /SIGPWR en un alpha pero SIGLOST en una sparc.)

Las letras en la columna "Acción" tienen los siguientes significados:

- A La acción por omisión es terminar el proceso.**
- B La acción por omisión es no hacer caso de la señal.**
- C La acción por omisión es terminar el proceso y hacer un volcado de memoria.**
- D La acción por omisión es parar el proceso.**
- E La señal no puede ser capturada.**
- F La señal no puede ser pasada por alto.**

CONFORME A POSIX.1

ERRORES

SIGIO y SIGLOST tienen el mismo valor. Este último está comentado en las fuentes del núcleo, pero el proceso de construcción de algunos programas aún piensa que la señal 29 es SIGLOST.

VÉASE TAMBIÉN

kill(1), kill(2), setitimer(2)

Linux 1.3.88

13 junio 1996

SIGNAL(7)

Veamos un ejemplo:

```
// Archivo: signals.c
#include <stdio.h>
#include <signal.h>
// Función: handler Una función que es static no se puede ver desde fuera de este programa
static void handler( int serial ){
    printf("Control - C!\n");
}
int main()
{
    signal( SIGINT , handler );
    for(;;);
    return 0; // salida exitosa del programa
}
// Fin del archivo: signals.c
```

Se compila con: **gcc -Wall signals.c -o signals.out**

Se ejecuta con: **./signals.out**

**Al ejecutar este programa se logra un proceso que es inmune a la interrupción por teclado
“Ctrl + C”**

¿Cómo hacemos para finalizar este proceso?

Debemos hacer uso del comando “ps”, para averiguar cuál es el **PID (PROCESS ID) del proceso**, que en este caso aparecerá listado como “**signals.out**”, de la siguiente manera:

ps aux | less

Nota: Esta es la forma en que se realiza un PIPE en la línea de comandos de GNU/Linux mediante “|”

Una vez que obtuvimos el PID debemos “**matar el proceso**” con el comando “**kill**” de la siguiente manera, **suponiendo que el PID es 4775**:

kill -QUIT 4775

Veamos otro ejemplo:

```
// Archivo: signals-2.c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
static void handler( int sig ){     printf("handler\n"); }
int main()
{
    alarm( 5 /* segundos */ );
    for( ; ); 
    return 0; // salida exitosa del programa
}
```

//Fin del archivo: signals-2.c

Se compila con: **gcc -Wall signals-2.c -o signals-2.out**

Se ejecuta con **./signals-2.out**

Al ejecutar este programa, el proceso espera unos 5 segundos, luego muestra en pantalla “**Temporizador**” y finaliza su ejecución.

Otro ejemplo:

```
// Archivo: signals-3.c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
static void handler( int sig ){ printf("handler!\n"); }

int main()
{
    signal( SIGALRM , handler );
    alarm( 5 ); // 5 segundos
    for(;;);
    return 0; // salida exitosa del programa
}

// Fin del archivo: signals-3.c
```

Se compila con: **gcc -Wall signals-3.c -o signals-3.out**

Se ejecuta con: **./signals-3.out**

Al ejecutar este programa, espera 5 segundos y entra en un bucle de repetición infinito.

Se puede finalizar el proceso mediante la interrupción por teclado “Ctrl + C”.

Ahora veamos otro ejemplo pero con multiproceso:

// Archivo: multiproceso.c

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>

int ya = 0; // variable global

static void handler( int sig )
{
    int status;
    wait( &status );
    ya = 1;
}

int main()
{
    if( fork() == 0 ) /* hijo */
    {
        printf( "Largamos!\n" );
        sleep(10);
        printf( "Chau!\n" );
        exit(0);
    }
    else /* padre */
    {
        signal( SIGCHLD , handler );
        while( ya == 0 )
            ;
        printf( "Al fin!\n" );
    }
    return 0;
}
// Fin del archivo: multiproceso.c
```

Se compila con: **gcc -Wall multiproceso.c -o multiproceso.out**

Se ejecuta con: **./multiproceso.out**

Al ejecutarlo, la salida en pantalla, luego de unos segundos, es:

Largamos!

Chau!

Al fin!

La semántica de signal:

Al volver de una señal, el “**handler**” hace un **return**, se vuelve a dónde se estaba al producirse la señal.

¿Se puede producir una señal estando dentro de una manipulador de otra señal previa?

Sí, aunque de otra categoría. Por lo general la señal activa queda bloqueada (así lo hacía BSD, el SYSTEM V de AT&T no). Un **manipulador asociado a una señal se mantiene entre invocaciones** (BSD así lo hacía, pero SYSTEM V de AT&T no).

¿Qué pasa si se genera la misma señal estando en el handler? ¿Se pierde?

Pues **no hay garantías**, aunque las primeras se activan al salir del manipulador (“handler”).

Donde **sí hay garantías de que no se pierden es con las señales de tiempo real (>= 32)**.

Existe un mecanismo alternativo mucho más detallado, usado por BSD.

En lugar de usar “**signal**” se usa otra función definida en **#include <signal.h>** llamada “**sigaction**”

La página del manual **man 2 sigaction** nos proporciona:

SIGACTION(2)

Manual del Programador de Linux

SIGACTION(2)

NOMBRE

sigaction, sigprocmask, sigpending, sigsuspend - funciones POSIX de manejo de señales

SINOPSIS

#include <signal.h>

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

```
int sigpending(sigset_t *set);
```

```
int sigsuspend(const sigset_t *mask);
```

DESCRIPCIÓN

La llamada al sistema **sigaction** se emplea para cambiar la acción tomada por un proceso cuando recibe una determinada señal.

signum especifica la señal y puede ser cualquiera válida salvo **SIGKILL** o **SIGSTOP**.

Si **act** no es nulo, la nueva acción para la señal **signum** se instala como **act**. Si **oldact** no es nulo, la acción anterior se guarda en **oldact**.

La estructura **sigaction** se define como

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
}
```

El elemento sa_restorer está obsoleto y no debería utilizarse. POSIX no especifica un elemento sa_restorer.

sa_handler especifica la acción que se va a asociar con signum y puede ser SIG_DFL para la acción predeterminada, SIG_IGN para no tener en cuenta la señal, o un puntero a una función manejadora para la señal.

sa_mask da una máscara de señales que deberían bloquearse durante la ejecución del manejador de señal. Además, la señal que lance el manejador será bloqueada, a menos que se activen las opciones SA_NODEFER o SA_NOMASK.

sa_flags especifica un conjunto de opciones que modifican el comportamiento del proceso de manejo de señal. Se forma por la aplicación del operador de bits OR a cero o más de las siguientes constantes:

SA_NOCLDSTOP

Si signum es SIGCHLD, no se reciba notificación cuando los procesos hijos se paren (esto es, cuando los procesos hijos reciban una de las señales SIGSTOP, SIGTSTP, SIGTTIN o SIGTTOU).

SA_ONESHOT o SA_RESETHAND

Restáurese la acción para la señal al estado predeterminado una vez que el manejador de señal haya sido llamado. (Esto es el comportamiento predeterminado de la llamada al sistema signal(2).)

SA_RESTART

Proporciona un comportamiento compatible con la semántica de señales de BSD haciendo re-ejecutables algunas llamadas al sistema entre señales.

SA_NOMASK o SA_NODEFER

No se impida que se reciba la señal desde su propio manejador.

SA_SIGINFO

El manejador de señal toma 3 argumentos, no uno. En este caso, se debe configurar sa_sigaction en lugar de sa_handler. (El campo sa_sigaction fue añadido en la versión 2.1.86 de Linux.)

El parámetro siginfo_t para sa_sigaction es una estructura con los siguientes elementos

```
siginfo_t {
    int    si_signo;          /* Número de señal */
    int    si_errno;          /* Un valor errno */
    int    si_code;           /* Código de señal */
    pid_t  si_pid;           /* ID del proceso emisor */
    uid_t  si_uid;           /* ID del usuario real del proceso emisor */
    int    si_status;         /* Valor de salida o señal */
    clock_t si_utime;        /* Tiempo de usuario consumido */
    clock_t si_stime;        /* Tiempo de sistema consumido */
    sigval_t si_value;       /* Valor de señal */
    int    si_int;            /* señal POSIX.1b */
    void * si_ptr;           /* señal POSIX.1b */
    void * si_addr;          /* Dirección de memoria que ha producido el fallo */
    int    si_band;           /* Evento de conjunto */
    int    si_fd;              /* Descriptor de fichero */
}
```

si_signo, si_errno y si_code están definidos para todas las señales. **kill(2)**, las señales **POSIX.1b** y **SIGCHLD** rellenan **si_pid** y **si_uid**. **SIGCHLD** también rellena **si_status**, **si_utime** y **si_stime**. **si_int** y **si_ptr** son especificados por el emisor de la señal **POSIX.1b**.

SIGILL, SIGFPE, SIGSEGV y SIGBUS rellenan **si_addr** con la dirección del fallo.
SIGPOLL rellena **si_band** y **si_fd**.

si_code indica por qué se ha enviado la señal. Es un valor, no una máscara de bits. Los valores que son posibles para cualquier señal se listan en la siguiente tabla:

si_code	
Valor	Origen de la señal
SI_USER	kill, sigsend o raise
SI_KERNEL	El núcleo
SI_QUEUE	sigqueue
SI_TIMER	el cronómetro ha vencido
SI_MESGQ	ha cambiado el estado de mesq
SI_ASYNCIO	ha terminado una E/S asíncrona
SI_SIGIO	SIGIO encolada

SIGILL	
ILL_ILLOPC	código de operación ilegal
ILL_ILLOPN	operando ilegal
ILL_ILLADR	modo de direccionamiento ilegal
ILL_ILLTRP	trampa ilegal
ILL_PRVOPC	código de operación privilegiada
ILL_PRVREG	registro privilegiado
ILL_COPROC	error del coprocesador
ILL_BADSTK	error de la pila interna

SIGFPE	
FPE_INTDIV	entero dividido por cero
FPE_INTOVF	desbordamiento de entero
FPE_FLTDIV	punto flotante dividido por cero
FPE_FLTOVF	desbordamiento de punto flotante
FPE_FLTUND	desbordamiento de punto flotante por defecto
FPE_FLTRES	resultado de punto flotante inexacto
FPE_FLTINV	operación de punto flotante inválida
FPE_FLTSUB	subscript fuera de rango

SIGSEGV	
SEGV_MAPERR	dirección no asociada a un objeto
SEGV_ACCERR	permisos inválidos para un objeto presente en memoria

SIGBUS	
BUS_ADRALN	alineamiento de dirección inválido
BUS_ADRERR	dirección física inexistente
BUS_OBJERR	error hardware específico del objeto

SIGTRAP	
TRAP_BRKPT	punto de parada de un proceso
TRAP_TRACE	trampa de seguimiento paso a paso de un proceso

SIGCHLD	
CLD_EXITED	ha terminado un hijo
CLD_KILLED	se ha matado a un hijo
CLD_DUMPED	un hijo ha terminado anormalmente
CLD_TRAPPED	un hijo con seguimiento paso a paso ha sido detenido
CLD_STOPPED	ha parado un hijo
CLD_CONTINUED	un hijo parado ha continuado

SIGPOLL	
POLL_IN	datos de entrada disponibles
POLL_OUT	buffers de salida disponibles
POLL_MSG	mensaje de entrada disponible
POLL_ERR	error de E/S
POLL_PRI	entrada de alta prioridad disponible
POLL_HUP	dispositivo desconectado

La llamada `sigprocmask` se emplea para cambiar la lista de señales bloqueadas actualmente. El comportamiento de la llamada depende del valor de `how`, como sigue:

SIG_BLOCK

El conjunto de señales bloqueadas es la unión del conjunto actual y el argumento `set`.

SIG_UNBLOCK

Las señales en `set` se quitan del conjunto actual de señales bloqueadas. Es legal intentar el desbloqueo de una señal que no está bloqueada.

SIG_SETMASK

El conjunto de señales bloqueadas se pone según el argumento `set`.

Si `oldset` no es nulo, el valor anterior de la máscara de señal se guarda en `oldset`.

La llamada `sigpending` permite el examen de señales pendientes (las que han sido producidas mientras estaban bloqueadas). La máscara de señal de las señales pendientes se guarda en `set`.

La llamada `sigsuspend` reemplaza temporalmente la máscara de señal para el proceso con la dada por `mask` y luego suspende el proceso hasta que se recibe una señal.

VALOR DEVUELTO

`sigaction`, `sigprocmask`, `sigpending` y `sigsuspend` devuelven 0 en caso de éxito y -1 en caso de error.

ERRORES

EINVAL Se ha especificado una señal inválida. Esto también se genera si se hace un intento de cambiar la acción para SIGKILL o SIGSTOP, que no pueden ser capturadas.

EFAULT `act`, `oldact`, `set` u `oldset` apuntan a una zona de memoria que no forma parte válida del espacio de direcciones del proceso.

EINTR La llamada al sistema ha sido interrumpida.

OBSERVACIONES

No es posible bloquear SIGKILL ni SIGSTOP con una llamada a `sigprocmask`. Los intentos de hacerlo no serán tenidos en cuenta, silenciosamente.

De acuerdo con POSIX, el comportamiento de un proceso está indefinido después de que no haga caso de una señal SIGFPE, SIGILL o SIGSEGV que no haya sido generada por las funciones `kill()` o `raise()`. La división entera por cero da un resultado indefinido. En algunas arquitecturas generará una señal SIGFPE. (También, el dividir el entero más negativo por -1 puede generar una señal SIGFPE.) No hacer caso de esta señal puede llevar a un bucle infinito.

POSIX (B.3.3.1.3) anula el establecimiento de SIG_IGN como acción para SIGCHLD. Los comportamientos de BSD y SYSV difieren, provocando el fallo en Linux de aquellos programas BSD que asignan SIG_IGN como acción para SIGCHLD.

La especificación POSIX sólo define `SA_NOCLDSTOP`. El empleo de otros valores en `sa_flags` no es transportable.

La opción `SA_RESETHAND` es compatible con la de SVr4 del mismo nombre.

La opción `SA_NODEFER` es compatible con la de SVr4 del mismo nombre bajo los núcleos 1.3.9 y posteriores. En núcleos más viejos la implementación de Linux permitía la recepción de cualquier señal, no sólo la que estábamos instalando (sustituyendo así efectivamente cualquier valor de `sa_mask`).

Los nombres `SA_RESETHAND` y `SA_NODEFER` para compatibilidad con SVr4 están presentes solamente en las versiones de la biblioteca 3.0.9 y mayores.

La opción `SA_SIGINFO` viene especificada por POSIX.1b. El soporte para ella se añadió en la versión 2.2 de Linux.

sigaction puede llamarse con un segundo argumento nulo para saber el manejador de señal en curso. También puede emplearse para comprobar si una señal dada es válida para la máquina donde se está, llamándola con el segundo y el tercer argumento nulos.

Vea `sigsetops(3)` para detalles sobre manipulación de conjuntos de señales.

CONFORME A

POSIX, SVr4. SVr4 no documenta la condición EINTR.

VÉASE TAMBIÉN

kill(1), kill(2), killpg(2), pause(2), raise(3), siginterrupt(3), signal(2), signal(7), sigsetops(3), sigvec(2)

Linux 2.2**8 mayo 1999****SIGACTION(2)**

Otras funciones útiles son “sigemptyset”, “sigfillset”, “sigaddset”, “sigdelset” y “sigismember” ver **man 3 sigsetops**, la cuál nos informa:

SIGSETOPS(3)**Manual del Programador de Linux****SIGSETOPS(3)****NOMBRE**

sigemptyset, sigfillset, sigaddset, sigdelset, sigismember - operaciones POSIX con conjuntos de señales

SINOPSIS

```
#include <signal.h>
```

```
int sigemptyset(sigset_t *conjunto);
int sigfillset(sigset_t *conjunto);
int sigaddset(sigset_t *conjunto, int numse);
int sigdelset(sigset_t *conjunto, int numse);
int sigismember(const sigset_t *conjunto, int numse);
```

DESCRIPCIÓN

La función **sigsetops(3)** permite la manipulación de conjuntos de señales, según la norma **POSIX**.

sigemptyset inicia el conjunto de señales dado por **conjunto** al conjunto vacío, con todas las señales fuera del conjunto.

sigfillset inicia conjunto al conjunto completo, con todas las señales incluidas en el conjunto.

sigaddset y **sigdelset** añaden y quitan respectivamente la señal **numse** de conjunto.

sigismember mira a ver si **numse** pertenece a conjunto.

VALOR DEVUELTO

sigemptyset, sigfullset, sigaddset y **sigdelset** devuelven 0 si acaban bien y -1 en caso de error.

sigismember devuelve 1 si **numse** es un miembro de conjunto, 0 si **numse** no lo es, y -1 en caso de error.

ERRORES

EINVAL **sig** no es una señal válida.

CONFORME A

POSIX

VÉASE TAMBIÉN

sigaction(2), sigpending(2), sigprocmask(2), sigsuspend(2)

Linux 1.0**9 Marzo 1998****SIGSETOPS(3)**

Veamos un ejemplo:

// Archivo: sigaction.c

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

static void handler( int sig ){ printf("Llega %d - Ja! Te agarré!\n", sig ); }

int main()
{
    struct sigaction sa;
    sa.sa_handler = handler;
    sigfillset( &sa.sa_mask );
    sa.sa_flags = SA_ONESHOT;
    sigaction( SIGINT , &sa , NULL );
    for(;;);
    return 0; // salida exitosa del programa
}

// Fin del archivo: sigaction.c
```

Se compila con: **gcc -Wall sigaction.c -o sigaction.out**

Se ejecuta con: **./sigaction.out**

Este programa al ejecutarlo **entra en un bucle de repetición infinito**, si se lo quiere interrumpir mediante el teclado con la combinación “**Ctrl + C**”, éste captura esa entrada y muestra en pantalla:

Llega 2 - Ja! Te agarré!

Si nuevamente se lo interrumpe con la misma combinación de teclado, el proceso finaliza.

Ahora comenzaremos una muy breve introducción al **Lenguaje de Programación Orientado a Objetos C++:**

Primero observemos como se realiza una **CLASE** en el otro **Lenguaje de Programación Orientado a Objetos “JAVA”:**

```
/* En JAVA la declaración de una CLASE */
class Q{      // Archivo: Q.java
    static void main( String[] args ){
        System.out.println("Hola Mundo!");
    }
}
```

Esta misma clase en C++:

```
/* En C++  Archivo: hola.cpp  */
#include <iostream>
using namespace std;
int main()
{
    cout << "Hola Mundo";
    return 0; // salida OK!
}
```

Los sufijos de los archivos de código fuente en Lenguaje C++ pueden ser:

- .cc
- .C
- .cpp

Un archivo fuente en C++ de compila con **g++** en forma similar a lo siguiente:

```
g++ fuente.cpp -o nombre.out
```

Se usa lo que se denomina “**namespace**”, es decir, “**espacio de nombres**” para **evitar ambigüedades** entre los nombres de los **ATRIBUTOS** de distintos grupos de **CLASES**. En este sentido los

“**namespaces**” de C++ son similares a los “**packages**” de JAVA.

Si no se emplea la expresión:

```
using namespace std;
```

entonces se deberán usar los **nombres totalmente cualificados de las CLASES**, como se muestra:

```
std::cout
```

C++ permite declarar variables antes de usarlas, por ejemplo:

```
// Archivo: segundo.cpp
#include <iostream>
int main()
{
    int i = 10;
    std::cout << i << "\n";
    int j = 20;
    std::cout << i+j << "\n";
    return 0; // salida OK!
}
// Fin Archivo: segundo.cpp
```

Se compila con: **g++ segundo.cpp -o segundo.out**

Se ejecuta con: **./segundo.out**

Su salida en pantalla es:

10

30

C++ permite declarar en las estructuras funciones (mientras que en C no se puede), por ejemplo:

```
// Archivo: funciones-dentro-de-estructuras.cc
#include <iostream>
struct S {
    int i;
    // declara una función dentro de la estructura
    void f(){ std::cout << i << "\n"; }
};
int main()
{
    // Para declarar una variable del tipo "struct S"
    // en C++ basta con poner: S nombre-variable
    // sin necesidad de poner struct.
    S a, b;
```

```
// Continúa Archivo: funciones-dentro-de-estrucutras.cc
    a.i = 10;// el operador "." es el selector de miembro de la estructura
    b.i = 20;
    a.f(); // aquí invoca a la función "f" de la "struct S" llamada "a"
    b.f();
    return 0; //salida OK!
}

// Fin Archivo: funciones-dentro-de-estrucutras.cc
```

Se compila con:

g++ -Wall funciones-dentro-de-estrucutras.cc -o funciones-dentro-de-estrucutras.out

Se ejecuta con: **./funciones-dentro-de-estrucutras.out**

Su salida en pantalla es: **10
20**

Las **estructuras en C++** pueden tener **CONSTRUCTORES**, veamos como se declara un constructor para la estructura que declaramos antes:

```
// Archivo: constructores-estructuras.cc
#include <iostream>

struct S{
    int i;
    /* Constructor de estructura S
     * S::i es el nombre totalmente cualificado
     *   de la variable entera "i"
     *   miembro de la estructura S
     */
    S( int i ){ S::i = i; };
    /* función f, miembro de la estructura S */
    int f(){ return i; }
};

int main()
{
    S a(10), b(20);
    std::cout << a.f() << ' ' << b.f() << "\n";
    return 0; // salida OK!
}
```

// Archivo: constructores-estructuras.cc

Se compila con: **g++ -Wall constructores-estructuras.cc -o constructores-estructuras.out**

Se ejecuta con: **./constructores-funciones.out**

Su salida en pantalla es: **10 20**

Las estructuras en C++ pueden tener DESTRUCTORES, estos son invocados cuando la variable (la INSTANCIA DE UNA CLASE) dejan de existir en memoria. Veamos un ejemplo:

```
// Archivo: destructores-estructuras.cc
#include <iostream>

struct S{
    int i;

    /* constructor */
    S( int i ){ S::i = i; std::cout << " NACE! " << S::i << "\n"; }

    /* destructor */
    ~S(){ std::cout << i << " EPPA!\n"; }
};

/* estructuras globales */
S a(1), b(2);

int main()
{
    S c(3), d(4);
    std::cout << " bloque! \n";

    /* bloque de código */
    {
        S e(5);
    }

    std::cout << " fin bloque!\n";
    return 0;      // salida OK!
}
```

// Fin Archivo: destructores-estructuras.cc

Se compila con: **g++ -Wall destructores-estructuras.cc -o destructores-estructuras.out**

Se ejecuta con: **./destructores-estructuras.out**

Su salida en pantalla es:

```
NACE! 1
NACE! 2
NACE! 3
NACE! 4
bloque!
NACE! 5
5 EPPA!
fin bloque!
4 EPPA!
3 EPPA!
2 EPPA!
1 EPPA!
```

Es evidente que las INSTANCIAS se DESTRUYEN en el orden inverso en que se CREARON.

Veamos una leve modificación del código fuente anterior en Lenguaje C++:

// Archivo: destructores-estructuras-2.cc

#include <iostream>

```
struct S{
    int i;

    /* constructor */
    S( int i ){ S::i = i; std::cout << " NACE! " << S::i << "\n"; }

    /* destructor */
    ~S(){ std::cout << i << " EPPA!\n"; }
};

/* estructuras globales */
S a(1), b(2);

int main()
{
    S c(3), d(4);
    std::cout << " bloque! \n";

    /* bloque de código */
    {
        static S e(5);      // Ahora declarado como static
    }

    std::cout << " fin bloque!\n";
    return 0;      // salida OK!
}
```

// Fin Archivo: destructores-estructuras-2.cc

Se compila con: **g++ -Wall destructores-estructuras-2.cc -o destructores-estructuras-2.out**

Se ejecuta con: **./destructores-estructuras-2.out**

Su salida en pantalla es:

```
NACE! 1
NACE! 2
NACE! 3
NACE! 4
bloque!
NACE! 5
fin bloque!
4 EPPA!
3 EPPA!
5 EPPA!
2 EPPA!
1 EPPA!
```

En C++ se usan punteros a objetos:

```
// Archivo: punteros-a-objetos.cc
#include <iostream>

struct S{
    int i;
    // constructor de estructura
    S( int i ){ S::i = i; std::cout << "NACE! " << i << " !\n"; }
    // destructor de estructura
    ~S(){ std::cout << "MUERE " << i << " !\n"; }
    void f(){ std::cout << ' ' << i << "\n"; }
};

int main()
{
    S a( 2 ), * pa = &a;
    a.f(); // usa la función mediante el nombre de la estructura
    pa->f(); // usa la función mediante el puntero a estructura
    return 0;
}
// Archivo: Fin punteros-a-objetos.cc
```

Se compila con: **g++ -Wall punteros-a-objetos.cc -o punteros-a-objetos.out**

Se ejecuta con: **./punteros-a-objetos.out**

Su salida en pantalla es:

NACE! 2 !

2

2

MUERE 2 !

En C++ también se permiten **miembros de estructuras compartidos**. Si declaramos:

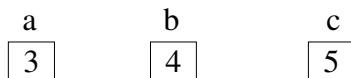
```
// Archivo: miembros-compartidos.cc
#include <iostream>

struct S{
    int i;
    S( int i ){ S::i = i; std::cout << "NACE " << i << " !\n"; }
    ~S(){ std::cout << "MUERE " << i << " !\n"; }
    void f(){ std::cout << i << "\n"; }
};

S a(3), b(4), c(5), * pa = &a, * pb = &b , * pc = &c;
// Tanto a, como b, como c, tienen su propio "int i"
int main()
{
    a.f();
    pa->f();
    b.f();
    pb->f();
    c.f();
    pc->f();
    return 0; //Salida OK!
} // Fin Archivo: miembros-compartidos.cc
```

Al compilarlo con: **g++ -Wall miembros-compartidos.cc -o miembros-compartidos.out** y ejecutarlo mediante: **./miembros-compartidos.out**

En memoria se obtendrá algo similar al siguiente esquema:



Y la salida en pantalla es:

NACE 3 !

NACE 4 !

NACE 5 !

3

3

4

4

5

5

MUERE 5 !

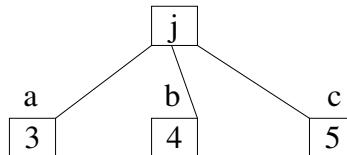
MUERE 4 !

MUERE 3 !

Pero si ahora declaramos:

```
struct S{
    int i;
    static int j;
}
```

Tendríamos este otro esquema:



Es más, **S::j** existe aún sin que hayan objetos de la estructura **S**.

En C++ también podemos declarar métodos estáticos: Por ejemplo:

```
struct S{
    int i;
    static int j;
    void static f(){ std::cout << j << "\n"; }
};
```

Un método estático sólo puede usar miembros estáticos.

Cuando tenemos:

a.f();	≡	S::f(a);	Ambos son accesibles por el nombre “this”
pa -> f()	≡	S::f(pa);	

Para las estructuras de C++ también tenemos reglas de accesibilidad, “scope”:

```
struct S{
    int i; // accesible en todo lugar
private:
    int j; // sólo accesible por los métodos de la estructura S
    int k; // sólo accesible por los métodos de la estructura S
public:
    int x; // accesible en todo lugar
};
```

Clases en C++:

Veamos que:

```
struct S{      // una estructura por defecto es public
    private:
        int i, j, k;
};
```

Es equivalente a:

```
class S{      // una clase es por defecto private
    int i,j,k;
}
```

Veamos otro ejemplo:

```
Class C{
    int i; // privado
public:
    int f( C c ){ return i + c.i; }
}
// ¿Es legal? Pues, sí!
```

C++ tiene referencias. Una referencia es un alias a un valor.

Ejemplo:

int i; int &ri = i; // “ri” es un alias de “i”, es decir, otro nombre para la variable “i”

Un alias permite simular el PASAJE POR REFERENCIA.

Ejemplo:

```
void swap( int &i , int &j )
{
    int tmp = i;
    i = j;
    j = tmp;
}
```

Se invoca en un “main”

```
int x, y;
swap( x, y );
```

¿Qué pasa si hacemos?

```
swap( 2 , 3 );
```

C++ hace:

```
{     int $i = 2, $j = 3, swap( $i , $j ); }
```

Crea las variables por nosotros y usa swap con esas variables.

C++ ofrece objetos dinámicos:

Se crean mediante new, que aloca memoria y llama al CONSTRUCTOR definido en la CLASE.

Se borran con delete, que invoca al DESTRUCTOR definido en la CLASE.

Ejemplo:

```
// Archivo: ejemplo-2.cc
#include <iostream>

class Q {
    int i;
    static int j;
public:
    /* constructor */
    Q( int i ){ j++; this->i = i ; std::cout << "NACE " << i << "!\n"; }
    /* destructor */
    ~Q(){ --j ; std::cout << "MUERE " << i << "!\n"; }
    static int cuantos(){ return j; }
};

int Q::j; /* Globales
Q a(1), b(2); */

int main()
{
    Q * p1 , * p2 , * p3;
    std::cout << Q::cuantos() << "\n"; p1 = new Q(3);
    std::cout << Q::cuantos() << "\n"; p2 = new Q(4);
    std::cout << Q::cuantos() << "\n"; p3 = new Q(5);
    std::cout << Q::cuantos() << "\n";
    std::cout << Q::cuantos() << "\n"; delete p1;
    std::cout << Q::cuantos() << "\n"; delete p2;
    std::cout << Q::cuantos() << "\n"; delete p3;

    return 0; // Salida OK!
} // Archivo: ejemplo-2.cc
```

Se compila con: **g++ -Wall ejemplo-2.cc -o ejemplo-2.out**

Se ejecuta con: **./ejemplo-2.out**

Su salida en pantalla es:

0

NACE 3!

1

NACE 4!

2

NACE 5!

3

3

MUERE 3!

2

MUERE 4!

1

MUERE 5!

Herencia y Clases derivadas en C++:

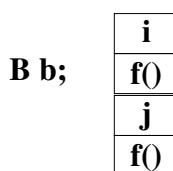
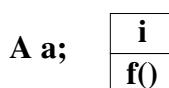
Vemos:

```
class A{
    int i;
public:
    int f() { return i; }
};
```

Para indicar que una CLASE B hereda de la CLASE A se utiliza el operador ":" como se muestra:

```
class B:A { // B hereda de A
    int j;
public:
    int f(){ return j + A::f(); }
};
```

De esta manera todo OBJETO de la CLASE B va a contener un OBJETO de la CLASE A, como se grafica a continuación:



Las CLASES derivadas no pueden usar los elementos private de sus SUPERCLASES, así como ésta tampoco puede acceder a los elementos public si no pone lo siguiente:

```
class B : public A{ ... } // B puede acceder a los ATRIBUTOS public de A
```

Vemos:

```
class A{
    int i; // private
public:
    A( int i ){ A::i = i; } /* constructor */
};

class B : public A { // B deriva de A y tiene acceso a los atributos public de A
    int i; // private
public:
    B( int i ) : A( i + 1 ) { B::i = i; }
    // A( i + 1 ) invoca al constructor de la CLASE A de la que hereda
};
```

Nota: Como todo OBJETO de la CLASE B va a contener un OBJETO de la CLASE A, donde se pida un A puede pasarse un B, y esto también tiene validez con los punteros.

```
// Archivo: herencia.cc
#include <iostream>
class A {
public:
    virtual void f() { std::cout << "A!\n"; }
};

class B : public A {
public:
    void f(){ std::cout << "B!\n"; }
};

void f( A * pa )
{
    pa -> f();
}

int main()
{
    A * pa = new A;
    B * pb = new B;
    f( pa );
    f( pb );
    return 0; // salida OK!
}
```

// Fin del Archivo: herencia.cc

Se compila con: **g++ -Wall herencia.cc -o herencia.out**

Se ejecuta con: **./herencia.out**

Su salida en pantalla es:
A!
B!

Clases ABSTRACTAS en C++:

En C++ una CLASE es ABSTRACTA si uno o más de sus MÉTODOS es ABSTRACTO.

Ejemplo:

```
class A{
public:
    void f() = 0 ; // un MÉTODO ABSTRACTO
}
```

NO SE PUEDEN CREAR OBJETOS DE UNA CLASE ABSTRACTA

Las CLASES ABSTRACTAS solo sirven para que otras CLASES HEREDEN a partir de la CLASE ABSTRACTA. Las CLASES DERIVADAS siguen siendo ABSTRACTAS excepto que redefinan TODOS los MÉTODOS ABSTRACTOS.

Ejemplo. Una calculadora que solo sumará expresiones:

```
// Archivo: calculadora.cc
#include <iostream>

class Expresion{      // CLASE ABSTRACTA
public:
    virtual int evaluar() = 0;
};      // ÚNICO MÉTODO ABSTRACTO

class Numero : public Expresion{      // CLASE CONCRETA
    int n; // privado
public:
    /* constructor */
    Numero( int n ):n(n){ }
    // inicializa también así: ":n(n)"
    int evaluar(){ return n; }
};

class Suma : public Expresion {
    Expresion * izq, * der; // privado
public:
    Suma( Expresion * izq , Expresion * der ):izq(izq),der(der){}
    int evaluar(){ return izq->evaluar() + der->evaluar(); }
};

int main()
{
    Expresion * expr =
        new Suma(
            new Numero(7) ,
            new Suma(
                new Numero(8) ,
                new Numero(9)
            )
        );
    std::cout << "Esto da " << expr->evaluar() << "\n";
    return 0;
}
// Fin Archivo: calculadora.cc
```

Se compila con: **g++ -Wall calculadora.cc -o calculadora.out**

Se ejecuta con: **./calculadora.out**

Su salida en pantalla es: **Esto da 24**

Arreglos dinámicos en C++:

Ejemplo:

```
// Archivo: arreglos-dinamicos.cc
#include <iostream>
using namespace std;

class C{
public:
    C(){ cout << "HOLA! \n"; } /* constructor */
    ~C(){ cout << "CHAU! \n"; } /* destructor */
};

int main()
{
    C * a;
    a = new C[5];
    delete[] a; // para arreglos
    return 0; // salida OK!
}

// Fin Archivo: arreglos-dinamicos.cc
```

Se compila con: **g++ -Wall arreglos-dinamicos.cc -o arreglos-dinamicos.out**

Se ejecuta con: **./arreglos-dinamicos.out**

Su salida en pantalla es:

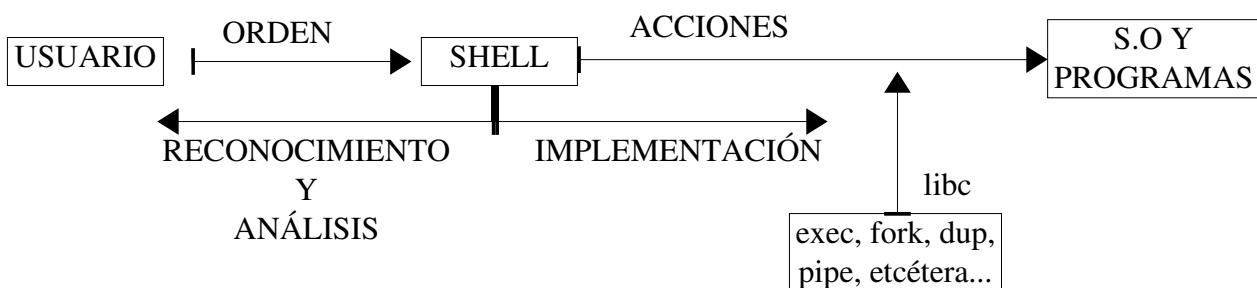
```
HOLA!
HOLA!
HOLA!
HOLA!
HOLA!
CHAU!
CHAU!
CHAU!
CHAU!
```

El Trabajo Final de la MATERIA:

Hay que hacer un minishell, que deberá incluir:

- salir con 'quit'
- ejecutar programas pasando argumentos. Los argumentos pueden tener comillas o se interpretarán como se muestra:
 - a b c son 3 argumentos
 - “a b c“ es un argumento
- deberá tener pipes
- deberá permitir redirección de entrada-salida:
 - programa > archivo sobre-escribe el archivo
 - programa >> archivo concatena, agrega al final del archivo
 - programa < archivo programa lee del archivo
- deberá tener condicionales
 - programa1 && programa2 programa2 se ejecuta si programa1 termina bien
 - programa1 || programa2 programa2 se ejecuta si programa1 termina mal
- deberá permitir sub-shells:
 - (programa) programa se ejecuta en otro shell
- deberá permitir secuencias:
 - programa1 ; programa2 1º se ejecuta programa1 y pase lo que pase con programa1 se ejecuta el programa2
- Es opcional que pueda utilizar el comando “cd” para cambiar de directorio

¿Qué significa hacer un shell?



Se comportaría como un intérprete. Para lograr esto debemos afrontar 3 etapas:

- Reconocimiento
- Análisis
- Implementación

Deberemos definir la gramática que deberán cumplir las órdenes y luego implementar una función que acepte y reconozca esa y sólo esa gramática.

Los programas que hacen esto se denominan **scanners** y **parsers**

Podemos generar estos programas a partir de una especificación.

Usaremos:

- flex para el escaner
- bison para el parser

Flex:

Flex toma una especificación de **EXPRESIONES REGULARES** y da como resultado un **AUTÓMATA FINITO**, un archivo de código fuente de un Lenguaje como C.

Bison:

Bison tomará una **GRAMÁTICA LIBRE DE CONTEXTO “LALR1”** y emitirá como resultado un **AUTÓMATA DE PILA**,

Jerarquía de Lenguajes según Norman Chomsky, 1954:

Lenguaje: es una relación entre símbolos (alfabeto)

Estas relaciones se expresan como pares (reglas)

A a B : c a a

A a B se desarma en c a a

Chomsky encontró que se puede definir 4 (cuatro) clases que se incluyen estrictamente y encontró que a cada uno de estos lenguajes le corresponde una máquina abstracta que la representa.

LENGUAJE	NOMBRE	MÁQUINA
0	Gramáticas Generales	Máquina de Turing
1	Gramáticas sensibles al contexto	Autómatas Lineales
2	Gramáticas libres de contexto	Autómatas de Pilas
3	Expresiones Regulares	Autómatas Finitos

Expresiones regulares:

Son una manera de describir conjuntos de “strings”. Dado un alfabeto $a_1, a_2, a_3, \dots, a_n$ una expresión regular se define como:

- un símbolo a_k es una expresión regular que denota al string formada por a_k
- Si e_1 y e_2 son expresiones regulares entonces:
 - e_1e_2 es una expresión regular que denota e_1e_2 , la **concatenación** de dos expresiones regulares es una expresión regular
 - $e_1|e_2$ es una expresión regular (alternativa) que denota e_1 ó e_2
 - e_1^* es una expresión regular (Repetición o estrella de Kleene) y denota $e(nada), e, ee, eee, e, e, e, \dots, e$

Esto alcanza pero por comodidad definiremos la **PRECEDENCIA**:

- **Mayor precedencia** tiene: **CONCATENACIÓN**
- le sigue: **REPETICIÓN**
- **Menor precedencia** tiene: **ALTERNATIVA**

Ejemplos: alfabeto: código ASCII

- Una **expresión regular que denota a la string 'quit'** , la solución es: **quit**
- Ahora hacemos **lo mismo pero a la manera de MS-DOS (NO IMPORTAN MAYÚSCULAS O MINÚSCULAS)**, la solución es: **(Q|q)(U|u)(I|i)(T|t)**
- Un **número decimal positivo cualquiera**, la solución es:

$$(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$$

Nota: En lugar de **(0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)**
se puede poner en forma abreviada así: **[0123456789]** o directamente así: **[0-9]**
Entonces para cualquier número decimal positivo la expresión regular sería también: **[0-9][0-9]***

Abreviaremos: **ee*** así: **e+**

Entonces el ejemplo del **número decimal positivo**, su **expresión regular** quedaría reducida a **[0-9]+**
Algunos ejemplos de lo que podría poner con lo obtenido en el ejemplo anterior:

- 0000 éste no es sensato
- 00017 éste es sensato

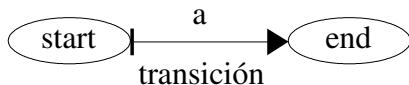
Para obtener sólo números sensatos, o sea, que no tengan CEROS a la izquierda, la expresión regular quedaría así: **0 | [1-9][0-9]***

Un **AUTÓMATA FINITO** está compuesto por un **conjunto finito de ESTADOS** y un **conjunto de TRANSICIONES** entre estados. Hay dos **estados distinguidos** llamados **start** y **end**.

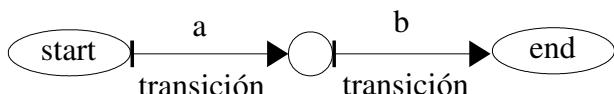
Si a cada **TRANSICIÓN** le asociamos un **SÍMBOLO DEL ALFABETO**, la correspondencia entre las **EXPRESIONES REGULARES** y los **AUTÓMATAS FINITOS** es evidente.

Ejemplos:

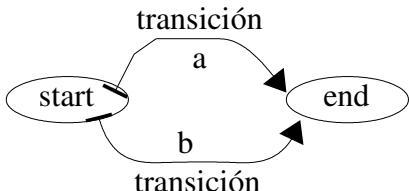
Expresión regular: a



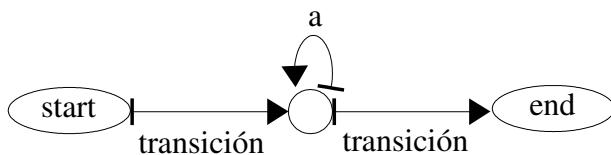
Concatenación: ab



Alternativa: a | b



Repetición: a*



Limitaciones de las expresiones regulares:

Lema de inyección(Pumping Lema):

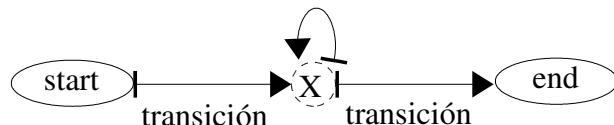
Decubiertas por Chomsky, significa que puede pasar de “start” a “end” vía los estados intermedios, si los hay, o sea,

Si un **AUTÓMATA FINITO** de **N ESTADOS** reconoce cadenas de largo **L** talque **0 < L, L>N**, entonces

estas cadenas serán de la forma: **C = AxB, con largo(x) > 0**

y reconocerá también: **Axx...xB , n > 0**

N veces

Esquema de la situación:

Veamos ahora algunos ejemplos:

- Hacer una expresión regular que reconozca solamente “strings” de este tipo: $((\dots(\)\dots))$
con K arbitrario

Aplicando el “**Lema de Pumping**” vemos que esto es imposible

Será:

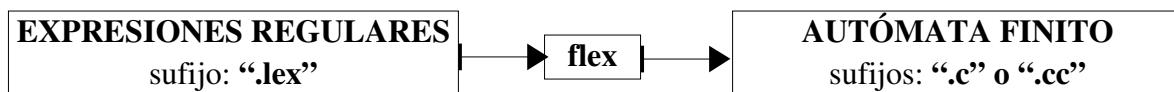
$$A = ((\dots(\ , \ B =)\dots))$$

$$X = ((\dots(\)\dots))$$

Por lo tanto **NO EXISTE ESTA EXPRESIÓN REGULAR. UNA EXPRESIÓN REGULAR NO PUEDE CONTAR PARÉNTESIS.**

Encontramos aquí una limitación en las expresiones regulares. Pese a esto vamos a ver **cómo crear un AUTÓMATA FINITO usando EXPRESIONES REGULARES.**

Esto se procesa con “flex”

Formato propio de un archivo “.lex”:

```

%{      // comienza código en Lenguaje C
        //Aquí puede poner código en Lenguaje C
%}      // finaliza código en Lenguaje C
// ABREVIATURAS Y CONTROLES
% %
//EXPRESIONES REGULARES ACCIONES PARA LAS EXPRESIONES REGULARES
// MÁS CÓDIGO EN LENGUAJE C
  
```

Veamos qué información nos proporciona la página del manual **man flex**:

FLEX(1)

FLEX(1)

NOMBRE

flex - generador de analizadores léxicos rápidos

SINOPSIS

flex [-bcdfhilnpstvwBFILTV78+? -C[aefFmr] -osalida -Pprefijo -Sesqueleto] [--help --version]
[nombrefichero ...]

INTRODUCCIÓN

Este manual describe **flex**, una herramienta para la generación de programas que realizan concordancia de patrones en texto. El manual incluye a la vez secciones de tutorial y de referencia:

Descripción

una breve introducción a la herramienta

Algunos Ejemplos Simples

Formato del Fichero de Entrada

Patrones

las expresiones regulares extendidas que utiliza flex

Cómo se Empareja la Entrada

las reglas para determinar lo que ha concordado

Acciones

cómo especificar qué hacer cuando concuerde un patrón

El Escáner Generado

detalles respecto al escáner que produce flex; cómo controlar la fuente de entrada

Condiciones de Arranque

la introducción de contexto en sus escáneres, y conseguir "mini-escáneres"

Múltiples Buffers de Entrada

cómo manipular varias fuentes de entrada; cómo analizar cadenas en lugar de ficheros.

Reglas de Fin-de-Fichero

reglas especiales para reconocer el final de la entrada

Macros Misceláneas

un sumario de macros disponibles para las acciones

Valores Disponibles para el Usuario

un sumario de valores disponibles para las acciones

Interfaz con Yacc

conectando escáneres de flex junto con analizadores de yacc

Opciones

opciones de línea de comando de flex, y la directiva "%option"

Consideraciones de Rendimiento

cómo hacer que sus analizadores vayan tan rápido como sea posible

Generando Escáneres en C++

la facilidad (experimental) para generar analizadores léxicos como clases de C++

Incompatibilidades con Lex y POSIX**cómo flex difiere del lex de AT&T y del lex estándar de POSIX****Diagnósticos**

esos mensajes de error producidos por flex (o por los escáneres que este genera) cuyo significado podría no ser evidente

Ficheros**los ficheros usados por flex****Deficiencias / Errores****problemas de flex conocidos****Ver También****otra documentación, herramientas relacionadas****Autor****incluye información de contacto****DESCRIPCIÓN**

flex es una herramienta para generar escáneres: programas que reconocen patrones léxicos en un texto. flex lee los ficheros de entrada dados, o la entrada estándar si no se le ha indicado ningún nombre de fichero, con la descripción de un escáner a generar. La descripción se encuentra en forma de parejas de expresiones regulares y código C, denominadas reglas. flex genera como salida un fichero fuente en C, lex.yy.c, que define una rutina yylex(). Este fichero se compila y se enlaza con la librería -lfl para producir un ejecutable. Cuando se arranca el fichero ejecutable, este analiza su entrada en busca de casos de las expresiones regulares. Siempre que encuentra uno, ejecuta el código C correspondiente.

ALGUNOS EJEMPLOS SIMPLES

En primer lugar veremos algunos ejemplos simples para una toma de contacto con el uso de flex. La siguiente entrada de flex especifica un escáner que siempre que encuentre la cadena "username" la reemplazará por el nombre de entrada al sistema del usuario:

```
% %
username printf( "%s", getlogin() );
```

Por defecto, cualquier texto que no reconozca el analizador léxico de flex se copia a la salida, así que el efecto neto de este escáner es copiar su fichero de entrada a la salida con cada aparición de "username" expandida. En esta entrada, hay solamente una regla. "username" es el patrón y el "printf" es la acción. El "%%" marca el comienzo de las reglas.

Aquí hay otro ejemplo simple:

```
int num_lineas = 0, num_caracteres = 0;
```

```
%%
\n    ++num_lineas; ++num_caracteres;
.    ++num_caracteres;

%%
main()
{
yylex();
printf( "# de líneas = %d, # de caracteres. = %d\n",
       num_lineas, num_caracteres );
}
```

Este analizador cuenta el número de caracteres y el número de líneas en su entrada (no produce otra salida que el informe final de la cuenta). La primera línea declara dos variables globales, "num_lineas" y "num_caracteres", que son visibles al mismo tiempo dentro de yylex() y en la rutina main() declarada después del segundo "% %". Hay dos reglas, una que empareja una línea nueva ("\n") e incrementa la cuenta de líneas y la cuenta de caracteres, y la que empareja cualquier carácter que no sea una línea nueva (indicado por la expresión regular ".").

Un ejemplo algo más complicado:

```
/* escáner para un lenguaje de juguete al estilo de Pascal */
```

```
%{
/* se necesita esto para la llamada a atof() más abajo */
#include <math.h>
%}
```

```
DIGITO [0-9]
ID    [a-z][a-z0-9]*
```

```
%%
{DIGITO}+ {
    printf( "Un entero: %s (%d)\n", yytext,
           atoi( yytext ) );
}

{DIGITO}+"."{DIGITO}* {
    printf( "Un real: %s (%g)\n", yytext,
           atof( yytext ) );
}

if|then|begin|end|procedure|function {
```

```
printf( "Una palabra clave: %s\n", yytext );
}

{ID}     printf( "Un identificador: %s\n", yytext );

"+"-/*"/" printf( "Un operador: %s\n", yytext );

"{}[^}\n]*"/* se come una linea de comentarios */

[ \t\n]+ /* se come los espacios en blanco */

.     printf( "Caracter no reconocido: %s\n", yytext );

%%

main( argc, argv )
int argc;
char **argv;
{
    ++argv, --argc; /* se salta el nombre del programa */
    if ( argc > 0 )
        yyin = fopen( argv[0], "r" );
    else
        yyin = stdin;

    yylex();
}
```

Esto podría ser los comienzos de un escáner simple para un lenguaje como Pascal. Este identifica diferentes tipos de tokens e informa a cerca de lo que ha visto.

Los detalles de este ejemplo se explicarán en las secciones siguientes.

FORMATO DEL FICHERO DE ENTRADA

El fichero de entrada de flex está compuesto de tres secciones, separadas por una línea donde aparece únicamente un %% en esta:

```
definiciones
%%
reglas
%%
código de usuario
```

La sección de definiciones contiene declaraciones de definiciones de nombres sencillas para simplificar la especificación del escáner, y declaraciones de condiciones de arranque, que se explicarán en una sección posterior.

Las definiciones de nombre tienen la forma:

nombre definición

El "nombre" es una palabra que comienza con una letra o un subrayado ('_') seguido por cero o más letras, dígitos, '_', o '-' (guión). La definición se considera que comienza en el primer carácter que no sea un espacio en blanco siguiendo al nombre y continuando hasta el final de la línea. Posteriormente se puede hacer referencia a la definición utilizando "{nombre}", que se expandirá a "(definición)". Por ejemplo,

DIGITO [0-9]

ID [a-z][a-zA-Z]*

define "DIGITO" como una expresión regular que empareja un dígito sencillo, e "ID" como una expresión regular que empareja una letra seguida por cero o más letras o dígitos. Una referencia posterior a

{DIGITO}+."{DIGITO}*

es idéntica a

([0-9])+."([0-9])*

y empareja uno o más dígitos seguido por un '.' seguido por cero o más dígitos.

La sección de reglas en la entrada de flex contiene una serie de reglas de la forma:

patrón acción

donde el patrón debe estar sin sangrar y la acción debe comenzar en la misma línea.

Ver más abajo para una descripción más amplia sobre patrones y acciones.

Finalmente, la sección de código de usuario simplemente se copia a lex.yy.c literalmente. Esta sección se utiliza para rutinas de complemento que llaman al escáner o son llamadas por este. La presencia de esta sección es opcional; Si se omite, el segundo %% en el fichero de entrada se podría omitir también.

En las secciones de definiciones y reglas, cualquier texto sangrado o encerrado entre %{} y %} se copia íntegramente a la salida (sin los %{}'s). Los %{}'s deben aparecer sin sangrar en líneas ocupadas únicamente por estos.

En la sección de reglas, cualquier texto o %{} sangrado que aparezca antes de la primera regla podría utilizarse para declarar variables que son locales a la rutina de análisis y (después de las declaraciones) al código que debe ejecutarse siempre que se entra a la rutina de análisis. Cualquier otro texto sangrado o %{} en la sección de reglas sigue copiándose a la salida, pero su significado no está bien definido y bien podría causar errores en tiempo de compilación (esta propiedad se presenta para conformidad con POSIX ; ver más abajo para otras características similares)

En la sección de definiciones (pero no en la sección de reglas), un comentario sin sangría (es decir, una línea comenzando con "/*") también se copia literalmente a la salida hasta el próximo "*/".

PATRONES

Los patrones en la entrada se escriben utilizando un conjunto extendido de expresiones regulares. Estas son:

x	empareja el carácter 'x'
.	cualquier carácter (byte) excepto una línea nueva
[xyz]	una "clase de caracteres"; en este caso, el patrón empareja una 'x', una 'y', o una 'z'
[abj-oZ]	una "clase de caracteres" con un rango; empareja una 'a', una 'b', cualquier letra desde la 'j' hasta la 'o', o una 'Z'
[^A-Z\n]	una "clase de caracteres negada", es decir, cualquier carácter menos los que aparecen en la clase. En este caso, cualquier carácter EXCEPTO una letra mayúscula.
r*	cualquier carácter EXCEPTO una letra mayúscula o una línea nueva
r+	una o más r's, donde r es cualquier expresión regular
r?	ceros o una r (es decir, "una r opcional")
r{2,5}	donde sea de dos a cinco r's
r{2,}	dos o más r's
r{4}	exactamente 4 r's
{nombre}	la expansión de la definición de "nombre" (ver más abajo)
"[xyz]\\"foo"	la cadena literal: [xyz]"foo
\X	si X es una 'a', 'b', 'f', 'n', 'r', 't', o 'v', entonces la interpretación ANSI-C de \x. En otro caso, un literal 'X' (usado para indicar operadores tales como '*')
\0	un carácter NUL (código ASCII 0)
\123	el carácter con valor octal 123
\x2a	el carácter con valor hexadecimal 2a
(r)	empareja una r; los paréntesis se utilizan para anular la precedencia (ver más abajo)
rs	la expresión regular r seguida por la expresión regular s; se denomina "concatenación"
r s	bien una r o una s
r/s	una r pero sólo si va seguida por una s. El texto emparejado por r se incluye cuando se determina si esta regla es el "emparejamiento"

más largo", pero se devuelve entonces a la entrada antes que se ejecute la acción. Así que la acción sólo ve el texto emparejado por r. Este tipo de patrones se llama "de contexto posterior".

(Hay algunas combinaciones de r/s que flex no puede emparejar correctamente; vea las notas en la sección Deficiencias / Errores más abajo respecto al "contexto posterior peligroso".)

^r una r, pero sólo al comienzo de una línea (es decir, justo al comienzo del análisis, o a la derecha después de que se haya analizado una línea nueva).

r\$ una r, pero sólo al final de una línea (es decir, justo antes de una línea nueva). Equivalente a "r\n".

Fíjese que la noción de flex de una "línea nueva" es exáctamente lo que el compilador de C utilizado para compilar flex interprete como '\n'; en particular, en algunos sistemas DOS debe filtrar los \r's de la entrada used mismo, o explícitamente usar r\r\n para "r\$".

<s>r una r, pero sólo en la condición de arranque s (ver más abajo para una discusión sobre las condiciones de arranque)

<s1,s2,s3>r lo mismo, pero en cualquiera de las condiciones de arranque s1, s2, o s3

<*>r una r en cualquier condición de arranque, incluso una exclusiva.

<<EOF>> un fin-de-fichero

<s1,s2><<EOF>> un fin-de-fichero en una condición de arranque s1 o s2

Fíjese que dentro de una clase de caracteres, todos los operadores de expresiones regulares pierden su significado especial excepto el carácter de escape ('\') y los operadores de clase de caracteres, '-'. Las expresiones regulares en el listado anterior están agrupadas de acuerdo a la precedencia, desde la precedencia más alta en la cabeza a la más baja al final. Aquellas agrupadas conjuntamente tienen la misma precedencia.

Por ejemplo,

foobar*

es lo mismo que

(foo)|(ba(r*))

ya que el operador '*' tiene mayor precedencia que la concatenación, y la concatenación más alta que el operador '|'. Este patrón por lo tanto empareja bien la cadena "foo" o la cadena "ba" seguida de cero o más r's. Para emparejar "foo" o, cero o más "bar"'s, use:

fool(bar)*

y para emparejar cero o más "foo"'s o "bar"'s:

(foolbar)*

Además de caracteres y rangos de caracteres, las clases de caracteres pueden también contener expresiones de clases de caracteres. Son expresiones encerradas entre los delimitadores [: y :] (que también deben aparecer entre el '[' y el ']' de la clase de caracteres; además pueden darse otros elementos dentro de la clase de caracteres). Las expresiones válidas son:

**[:alnum:] [:alpha:] [:blank:]
[:cntrl:] [:digit:] [:graph:]
[:lower:] [:print:] [:punct:]
[:space:] [:upper:] [:xdigit:]**

Todas estas expresiones designan un conjunto de caracteres equivalentes a la correspondiente función estándar isXXX de C. Por ejemplo, [:alnum:] designa aquellos caracteres para los cuales isalnum() devuelve verdadero - esto es, cualquier carácter alfabético o numérico. Algunos sistemas no ofrecen isblank(), así que flex define [:blank:] como un espacio en blanco o un tabulador.

Por ejemplo, las siguientes clases de caracteres son todas equivalentes:

**[[:alnum:]]
[[:alpha:][:digit:]]
[[:alpha:]0-9]
[a-zA-Z0-9]**

Si su escáner ignora la distinción entre mayúsculas y minúsculas (la bandera -i), entonces [:upper:] y [:lower:] son equivalentes a [:alpha:].

Algunas notas sobre los patrones:

- Una clase de caracteres negada tal como el ejemplo "[^A-Z]" anterior emparejará una línea nueva a menos que "\n" (o una secuencia de escape equivalente) sea uno de los caracteres presentes explícitamente en la clase de caracteres negada (p.ej., "[^A-Z\n]"). Esto es diferente a cómo muchas de las otras herramientas de expresiones regulares tratan las clases de caracteres negadas, pero desafortunadamente la inconsistencia está fervientemente enraizada históricamente. Emparejar líneas nuevas significa que un patrón como [""]* puede emparejar la entrada completa a menos que haya otra comilla en la entrada.

- Una regla puede tener lo más una instancia del contexto posterior (el operador '/' o el

operador '\$'). La condición de arranque, los patrones '^', y "<<EOF>>" pueden aparecer solamente al principio de un patrón, y, al igual que con '/' y '\$', no pueden agruparse dentro de paréntesis. Un '^' que no aparezca al principio de una regla o un '\$' que no aparezca al final de una regla pierde sus propiedades especiales y es tratado como un carácter normal.

Lo siguiente no está permitido:

```
foo/bar$  
<sc1>foo<sc2>bar
```

Fíjese que la primera regla se puede escribir como "foo/bar\n".

En el siguiente ejemplo un '\$' o un '^' es tratado como un carácter normal:

```
fool(bar$)  
fool^bar
```

Si lo que se desea es un "foo" o un "bar" seguido de una línea nueva, puede usarse lo siguiente (la acción especial '\l' se explica más abajo):

```
foo    |  
bar$  /* la acción va aquí */
```

Un truco parecido funcionará para emparejar un "foo" o, un "bar" al principio de una línea.

CÓMO SE EMPAREJA LA ENTRADA

Cuando el escáner generado está funcionando, este analiza su entrada buscando cadenas que concuerden con cualquiera de sus patrones. Si encuentra más de un emparejamiento, toma el que empareje más texto (para reglas de contexto posterior, se incluye la longitud de la parte posterior, incluso si se devuelve a la entrada). Si encuentra dos o más emparejamientos de la misma longitud, se escoge la regla listada en primer lugar en el fichero de entrada de flex.

Una vez que se determina el emparejamiento, el texto correspondiente al emparejamiento (denominado el token) está disponible en el puntero a carácter global `yytext`, y su longitud en la variable global entera `yylen`.

Entonces la acción correspondiente al patrón emparejado se ejecuta (una descripción más detallada de las acciones viene a continuación), y entonces la entrada restante se analiza para otro emparejamiento.

Si no se encuentra un emparejamiento, entonces se ejecuta la regla por defecto: el siguiente carácter en la entrada se considera reconocido y se copia a la salida estándar. Así, la entrada válida más simple de flex es:

```
% %
```

que genera un escáner que simplemente copia su entrada (un carácter a la vez) a la salida.

Fíjese que `yytext` se puede definir de dos maneras diferentes: bien como un puntero a

caracter o como un array de caracteres. Usted puede controlar la definición que usa **flex** incluyendo una de las directivas especiales **%pointer** o **%array** en la primera sección (definiciones) de su entrada de flex. Por defecto es **%pointer**, a menos que use la opción de compatibilidad **-l**, en cuyo caso **yytext** será un **array**. La ventaja de usar **%pointer** es un análisis substancialmente más rápido y la ausencia de desbordamiento del buffer cuando se emparejen tokens muy grandes (a menos que se agote la memoria dinámica). La desventaja es que se encuentra restringido en cómo sus acciones pueden modificar **yytext** (vea la siguiente sección), y las llamadas a la función **unput()** destruyen el contenido actual de **yytext**, que puede convertirse en un considerable quebradero de cabeza de portabilidad al cambiar entre diferentes versiones de **lex**.

La ventaja de **%array** es que entonces puede modificar **yytext** todo lo que usted quiera, las llamadas a **unput()** no destruyen **yytext** (ver más abajo). Además, los programas de **lex** existentes a veces acceden a **yytext** externamente utilizando declaraciones de la forma:

```
extern char yytext[];
```

Esta definición es errónea cuando se utiliza **%pointer**, pero correcta para **%array**.

%array define a **yytext** como un array de **YYLMAX** caracteres, que por defecto es un valor bastante grande. Usted puede cambiar el tamaño simplemente definiendo con **#define** a **YYLMAX** con un valor diferente en la primera sección de su entrada de **flex**. Como se mencionó antes, con **%pointer** **yytext** crece dinámicamente para acomodar tokens grandes. Aunque esto signifique que con **%pointer** su escáner puede acomodar tokens muy grandes (tales como emparejar bloques enteros de comentarios), tenga presente que cada vez que el escáner deba cambiar el tamaño de **yytext** también debe reiniciar el análisis del token entero desde el principio, así que emparejar tales tokens puede resultar lento. Ahora **yytext** no crece dinámicamente si una llamada a **unput()** hace que se deba devolver demasiado texto; en su lugar, se produce un error en tiempo de ejecución.

También tenga en cuenta que no puede usar **%array** en los analizadores generados como clases de C++ (la opción **c++**; vea más abajo).

ACCIONES

Cada patrón en una regla tiene una acción asociada, que puede ser cualquier sentencia en C. El patrón finaliza en el primer carácter de espacio en blanco que no sea una secuencia de escape; lo que queda de la línea es su acción. Si la acción está vacía, entonces cuando el patrón se empareje el token de entrada simplemente se descarta. Por ejemplo, aquí está la especificación de un programa que borra todas las apariciones de "zap me" en su entrada:

```
%%
"zap me"
```

(Este copiará el resto de caracteres de la entrada a la salida ya que serán emparejados por la regla por defecto.)

Aquí hay un programa que comprime varios espacios en blanco y tabuladores a un solo espacio en blanco, y desecha los espacios que se encuentren al final de una línea:

```
%%
[ \t]+    putchar( ' ' );
[ \t]+$   /* ignora este token */
```

Si la acción contiene un '{', entonces la acción abarca hasta que se encuentre el correspondiente '}', y la acción podría entonces cruzar varias líneas. flex es capaz de reconocer las cadenas y comentarios de C y no se dejará engañar por las llaves que encuentre dentro de estos, pero aun así también permite que las acciones comiencen con %{ y considerará que la acción es todo el texto hasta el siguiente %} (sin tener en cuenta las llaves ordinarias dentro de la acción).

Una acción que consista sólamente de una barra vertical ('|') significa "lo mismo que la acción para la siguiente regla." Vea más abajo para una ilustración.

Las acciones pueden incluir código C arbitrario, incuyendo sentencias return para devolver un valor desde cualquier rutina llamada yylex(). Cada vez que se llama a yylex() esta continúa procesando tokens desde donde lo dejó la última vez hasta que o bien llegue al final del fichero o ejecute un return.

Las acciones tienen libertad para modificar yytext excepto para alargarla (añadiendo caracteres al final--esto sobreescibirá más tarde caracteres en el flujo de entrada). Sin embargo esto no se aplica cuando se utiliza %array (ver arriba); en ese caso, yytext podría modificarse libremente de cualquier manera.

Las acciones tienen libertad para modificar yyleng excepto que estas no deberían hacerlo si la acción también incluye el uso de yymore() (ver más abajo).

Hay un número de directivas especiales que pueden incluirse dentro de una acción:

- **ECHO copia yytext a la salida del escáner.**
- **BEGIN seguido del nombre de la condición de arranque pone al escáner en la condición de arranque correspondiente (ver más abajo).**
- **REJECT ordena al escáner a que proceda con la "segunda mejor" regla que concuerde con la entrada (o un prefijo de la entrada). La regla se escoge como se describió anteriormente en "Cómo se Empareja la Entrada", y yytext e yyleng se ajustan de forma apropiada. Podría ser una que empareje tanto texto como la regla escogida originalmente pero que viene más tarde en el fichero de entrada de flex, o una que empareje menos texto. Por ejemplo, lo que viene a continuación contará las palabras en la entrada y llamará a la rutina especial() siempre que vea "frob":**

```
int contador_palabras = 0;  
%  
  
frob    especial(); REJECT;  
[^ \t\n]+  ++contador_palabras;
```

Sin el REJECT, cualquier número de "frob"'s en la entrada no serían contados como palabras, ya que el escáner normalmente ejecuta solo una acción por token. Se permite el uso de múltiples REJECT's, cada uno buscando la siguiente mejor elección a la regla que actualmente esté activa. Por ejemplo, cuando el siguiente escáner analice el token "abcd", este escribirá "abcdabcaba" a la salida:

```
% %
a   |
ab  |
abc |
abcd ECHO; REJECT;
.\n /* se come caracteres sin emparejar */
```

(Las primeras tres reglas comparten la acción de la cuarta ya que estas usan la acción especial '|'.)

REJECT es una propiedad particularmente cara en términos de rendimiento del escáner; si se usa en cualquiera de las acciones del escáner esta ralentizará todo el proceso de emparejamiento del escáner.

Además, **REJECT** no puede usarse con las opciones **-Cf** o **-CF** (ver más abajo).

Fíjese también que a diferencia de las otras acciones especiales, **REJECT** es una bifurcación; el código que la siga inmediatamente en la acción no será ejecutado.

- **yymore()** dice al escáner que la próxima vez que empareje una regla, el token correspondiente debe ser añadido tras el valor actual de **yytext** en lugar de reemplazarlo. Por ejemplo, dada la entrada "mega-klugde" lo que viene a continuación escribirá "mega-mega-kludge" a la salida:

```
% %
mega- ECHO; yymore();
kludge ECHO;
```

El primer "mega-" se empareja y se repite a la salida. Entonces se empareja "kludge", pero el "mega-" previo aún está esperando al inicio de **yytext** así que el **ECHO** para la regla del "kludge" realmente escribirá "mega-kludge".

Dos notas respecto al uso de **yymore()**. Primero, **yymore()** depende de que el valor de **yy leng** refleje correctamente el tamaño del token actual, así que no debe modificar **yy leng** si está utilizando **yymore()**. Segundo, la presencia de **yymore()** en la acción del escáner implica una pequeña penalización de rendimiento en la velocidad de emparejamiento del escáner.

- **yyless(n)** devuelve todos excepto los primeros n caracteres del token actual de nuevo al flujo de entrada, donde serán reanalizados cuando el escáner busque el siguiente emparejamiento. **yytext** e **yy leng** se ajustan de forma adecuada (p.ej., **yy leng** no será igual a n). Por ejemplo, con la entrada "foobar" lo que viene a continuación escribirá "foobarbar":

```
% %
foobar ECHO; yyless(3);
[a-z]+ ECHO;
```

Un argumento de 0 para **yyless** hará que la cadena de entrada actual sea analizada por completo de nuevo.

A menos que haya cambiado la manera en la que el escáner procese de ahora en adelante su entrada (utilizando **BEGIN**, por ejemplo), esto producirá un bucle sin fin.

Fíjese que **yyless** es una macro y puede ser utilizada solamente en el fichero de entrada de flex, no desde otros ficheros fuente.

- **unput(c)** pone el carácter c de nuevo en el flujo de entrada. Este será el próximo carácter analizado.

La siguiente acción tomará el token actual y hará que se vuelva a analizar pero encerrado entre paréntesis.

```
{
int i;
/* Copia yytext porque unput() desecha yytext */
char *yycopia = strdup( yytext );
unput( ')' );
for ( i = yyleng - 1; i >= 0; --i )
    unput( yycopia[i] );
unput( '(' );
free( yycopia );
}
```

Fíjese que ya que cada **unput()** pone el carácter dado de nuevo al principio del flujo de entrada, al devolver cadenas de caracteres se debe hacer de atrás hacia delante.

Un problema potencial importante cuando se utiliza **unput()** es que si está usando %pointer (por defecto), una llamada a **unput()** destruye el contenido de **yytext**, comenzando con su carácter más a la derecha y devorando un carácter a la izquierda con cada llamada. Si necesita que se preserve el valor de **yytext** después de una llamada a **unput()** (como en el ejemplo anterior), usted debe o bien copiarlo primero en cualquier lugar, o construir su escáner usando %array (ver Cómo se Empareja la Entrada).

Finalmente, note que no puede devolver EOF para intentar marcar el flujo de entrada con un fin-de-fichero.

- **input()** lee el próximo carácter del flujo de entrada. Por ejemplo, lo que viene a continuación es una manera de comerse los comentarios en C:

```
% %
"/**"
{
register int c;

for ( ; ; )
{
    while ( (c = input()) != '*' &&
           c != EOF )
        ; /* se come el texto del comentario */

    if ( c == '*' )
    {
        while ( (c = input()) == '*' )
            ;
        if ( c == '/' )
            break; /* encontró el final */
    }
}
```

```
if ( c == EOF )
{
    error( "EOF en comentario" );
    break;
}
}
```

(Fíjese que si el escáner se compila usando C++, entonces a `input()` se le hace referencia con `yyinput()`, para evitar una colisión de nombre con el flujo de C++ por el nombre `input()`.)

- `YY_FLUSH_BUFFER` vacía el buffer interno del escáner de manera que la próxima vez que el escáner intente emparejar un token, este primero rellenará el buffer usando `YY_INPUT` (ver El Escáner Generado, más abajo). Esta acción es un caso especial de la función más general `yy_flush_buffer()`, descrita más abajo en la sección Múltiples Buffers de Entrada.
- `yyterminate()` se puede utilizar en lugar de una sentencia de retorno en una acción. Esta hace que finalice el escáner y retorne un 0 a quien haya llamado al escáner, indicando que "todo está hecho". Por defecto, también se llama a `yyterminate()` cuando se encuentra un fin-de-fichero. Esta es una macro y podría ser redefinida.

El Escáner Generado

La salida de flex es el fichero `lex.yy.c`, que contiene la rutina de análisis `yylex()`, un número de tablas usadas por esta para emparejar tokens, y un número de rutinas auxiliares y macros. Por defecto, `yylex()` se declara así

```
int yylex()
{
    ... aquí van varias definiciones y las acciones ...
}
```

(Si su entorno acepta prototipos de funciones, entonces este será "int `yylex(void)`"). Esta definición podría modificarse definiendo la macro "`YY_DECL`". Por ejemplo, podría utilizar:

```
#define YY_DECL float lexscan( a, b ) float a, b;
```

para darle a la rutina de análisis el nombre `lexscan`, que devuelve un real, y toma dos reales como argumentos.

Fíjese que si pone argumentos a la rutina de análisis usando una declaración de función no-prototipada/tipo-K&R, debe hacer terminar la definición con un punto y coma (;).

Siempre que se llame a `yylex()`, este analiza tokens desde el fichero de entrada global `yyin` (que por defecto es igual a `stdin`). La función continúa hasta que alcance el final del fichero (punto en el que devuelve el valor 0) o una de sus acciones ejecute una sentencia `return`.

Si el escáner alcanza un fin-de-fichero, entonces el comportamiento en las llamadas posteriores está indefinido a menos que o bien `yyin` apunte a un nuevo fichero de entrada (en cuyo caso el análisis continúa a partir de ese fichero), o se llame a `yyrestart()`. `yyrestart()` toma un argumento, un puntero `FILE *` (que puede ser nulo, si ha preparado a `YY_INPUT` para que analice una fuente distinta a `yyin`), e inicializa `yyin` para que escanee ese fichero.

Esencialmente no hay diferencia entre la asignación a yyin de un nuevo fichero de entrada o el uso de yyrestart() para hacerlo; esto último está disponible por compatibilidad con versiones anteriores de flex, y porque puede utilizarse para conmutar ficheros de entrada en medio del análisis. También se puede utilizar para desechar el buffer de entrada actual, invocándola con un argumento igual a yyin; pero mejor es usar YY_FLUSH_BUFFER (ver más arriba). Fíjese que yyrestart() no reinicializa la condición de arranque a INITIAL (ver Condiciones de Arranque, más abajo).

Si yylex() para el análisis debido a la ejecución de una sentencia return en una de las acciones, el analizador podría ser llamado de nuevo y este reanudaría el análisis donde lo dejó.

Por defecto (y por razones de eficiencia), el analizador usa lecturas por bloques en lugar de simples llamadas a getc() para leer caracteres desde yyin. La manera en la que toma su entrada se puede controlar definiendo la macro YY_INPUT. La secuencia de llamada para YY_INPUT es "YY_INPUT(buf,result,max_size)". Su acción es poner hasta max_size caracteres en el array de caracteres buf y devolver en la variable entera result bien o el número de caracteres leídos o la constante YY_NULL (0 en sistemas Unix) para indicar EOF. Por defecto YY_INPUT lee desde la variable global puntero a fichero "yyin".

Una definición de ejemplo para YY_INPUT (en la sección de definiciones del fichero de entrada) es:

```
%{  
#define YY_INPUT(buf,result,max_size) \  
{ \  
int c = getchar(); \  
result = (c == EOF) ? YY_NULL : (buf[0] = c, 1); \  
}  
}%
```

Esta definición cambiará el procesamiento de la entrada para que suceda un carácter a la vez.

Cuando el analizador reciba una indicación de fin-de-fichero desde YY_INPUT, entonces esta comprueba la función yywrap(). Si yywrap() devuelve falso (cero), entonces se asume que la función ha ido más allá y ha preparado yyin para que apunte a otro fichero de entrada, y el análisis continúa. Si este retorna verdadero (no-cero), entonces el analizador termina, devolviendo un 0 a su invocador. Fíjese que en cualquier caso, la condición de arranque permanece sin cambios; esta no vuelve a ser INITIAL.

Si no proporciona su propia versión de yywrap(), entonces debe bien o usar %option noyywrap (en cuyo caso el analizador se comporta como si yywrap() devolviera un 1), o debe enlazar con -lfl para obtener la versión por defecto de la rutina, que siempre devuelve un 1.

Hay disponibles tres rutinas para analizar desde buffers de memoria en lugar de desde ficheros: yy_scan_string(), yy_scan_bytes(), e yy_scan_buffer(). Las trataremos en la sección Múltiples Buffers de Entrada.

El analizador escribe su salida con ECHO a la variable global yyout (por defecto, stdout), que el usuario podría redefinir asignándole cualquier otro puntero a FILE.

CONDICIONES DE ARRANQUE

flex dispone de un mecanismo para activar reglas condicionalmente. Cualquier regla cuyo patrón se prefije con "<sc>" únicamente estará activa cuando el analizador se encuentre en la

condición de arranque llamada "sc". Por ejemplo,

```
<STRING>[^']* { /* se come el cuerpo de la cadena ... */  
...  
}
```

estará activa solamente cuando el analizador esté en la condición de arranque "STRING", y

```
<INITIAL,STRING,QUOTE>\. { /* trata una secuencia de escape ... */  
...  
}
```

estará activa solamente cuando la condición de arranque actual sea o bien "INITIAL", "STRING", o "QUOTE".

Las condiciones de arranque se declaran en la (primera) sección de definiciones de la entrada usando líneas sin sangrar comenzando con %s o %x seguida por una lista de nombres. Lo primero declara condiciones de arranque inclusivas, lo último condiciones de arranque exclusivas. Una condición de arranque se activa utilizando la acción BEGIN. Hasta que se ejecute la próxima acción BEGIN, las reglas con la condición de arranque dada estarán activas y las reglas con otras condiciones de arranque estarán inactivas. Si la condición de arranque es inclusiva, entonces las reglas sin condiciones de arranque también estarán activas. Si es exclusiva, entonces sólamente las reglas calificadas con la condición de arranque estarán activas. Un conjunto de reglas dependientes de la misma condición de arranque exclusiva describe un analizador que es independiente de cualquiera de las otras reglas en la entrada de flex. Debido a esto, las condiciones de arranque exclusivas hacen fácil la especificación de "mini-escáneres" que analizan porciones de la entrada que son sintácticamente diferentes al resto (p.ej., comentarios).

Si la distinción entre condiciones de arranque inclusivas o exclusivas es aún un poco vaga, aquí hay un ejemplo simple que ilustra la conexión entre las dos. El conjunto de reglas:

```
%s ejemplo  
%%
```

```
<ejemplo>foo hacer_algo();
```

```
bar algo_mas();
```

es equivalente a

```
%x ejemplo  
%%
```

```
<ejemplo>foo hacer_algo();
```

```
<INITIAL,example>bar algo_mas();
```

Sin el calificador <INITIAL,example>, el patrón bar en el segundo ejemplo no estará activo (es decir, no puede emparejarse) cuando se encuentre en la condición de arranque example.

Si hemos usado `<example>` para calificar bar, aunque, entonces este únicamente estará activo en example y no en INITIAL, mientras que en el primer ejemplo está activo en ambas, porque en el primer ejemplo la condición de arranque example es una condición de arranque inclusiva (%s).

Fíjese también que el especificador especial de la condición de arranque `<*>` empareja todas las condiciones de arranque. Así, el ejemplo anterior también pudo haberse escrito;

```
%x ejemplo
%%
```

```
<ejemplo>foo  hacer_algo();
```

```
<*>bar  algo_mas();
```

La regla por defecto (hacer un ECHO con cualquier carácter sin emparejar) permanece activa en las condiciones de arranque. Esta es equivalente a:

```
<*>.\n  ECHO;
```

`BEGIN(0)` retorna al estado original donde solo las reglas sin condiciones de arranque están activas. Este estado también puede referirse a la condición de arranque "INITIAL", así que `BEGIN(INITIAL)` es equivalente a `BEGIN(0)`. (No se requieren los paréntesis alrededor del nombre de la condición de arranque pero se considera de buen estilo.)

Las acciones `BEGIN` pueden darse también como código sangrado al comienzo de la sección de reglas. Por ejemplo, lo que viene a continuación hará que el analizador entre en la condición de arranque "ESPECIAL" siempre que se llame a `yylex()` y la variable global `entra_en_especial` sea verdadera:

```
int entra_en_especial;
```

```
%x ESPECIAL
%%
if ( entra_en_especial )
  BEGIN(ESPECIAL);
```

```
<ESPECIAL>blablabla
...más reglas a continuación...
```

Para ilustrar los usos de las condiciones de arranque, aquí hay un analizador que ofrece dos interpretaciones diferentes para una cadena como "123.456". Por defecto este la tratará como tres tokens, el entero "123", un punto ('.'), y el entero "456". Pero si la cadena viene precedida en la línea por la cadena "espera-reales" este la tratará como un único token, el número en coma flotante 123.456:

```
%{
#include <math.h>
%
%s espera
```

```
%%
espera-reales      BEGIN(espera);

<espera>[0-9]+."[0-9]+  {
    printf( "encontró un real, = %f\n",
            atof( yytext ) );
}

<espera>\n      {
    /* este es el final de la línea,
     * así que necesitamos otro
     * "espera-numero" antes de
     * que volvamos a reconocer más
     * números
     */
    BEGIN(INITIAL);
}

[0-9]+  {
    printf( "encontró un entero, = %d\n",
            atoi( yytext ) );
}

"."    printf( "encontró un punto\n" );
```

Aquí está un analizador que reconoce (y descarta) comentarios de C mientras mantiene una cuenta de la línea actual de entrada.

```
%x comentario
%%
int num_linea = 1;

"/*"      BEGIN(comentario);

<comentario>[^*\n]*  /* come todo lo que no sea '*' */
<comentario>"*"+[^*/\n]* /* come '*'s no seguidos por '/' */
<comentario>\n      ++num_linea;
<comentario>"*"+"/"    BEGIN(INITIAL);
```

Este analizador se complica un poco para emparejar tanto texto como le sea posible en cada regla. En general, cuando se intenta escribir un analizador de alta velocidad haga que cada regla empareje lo más que pueda, ya que esto es un buen logro.

Fíjese que los nombres de las condiciones de arranque son realmente valores enteros y pueden ser almacenados como tales. Así, lo anterior podría extenderse de la siguiente manera:

```
%x comentario foo
%%
int num_linea = 1;
```

```
int invocador_comentario;
```

```
""/*" {  
    invocador_comentario = INITIAL;  
    BEGIN(comentario);  
}
```

...

```
<foo>""/*" {  
    invocador_comentario = foo;  
    BEGIN(comentario);  
}
```

```
<comentario>[^*\n]* /* se come cualquier cosa que no sea un '*' */  
<comentario>"""+[^*\n]* /* se come '*'s que no continuen con '/'s */  
<comentario>\n      ++num_linea;  
<comentario>"""+"/"   BEGIN(invocador_comentario);
```

Además, puede acceder a la condición de arranque actual usando la macro de valor entero YY_START. Por ejemplo, las asignaciones anteriores a invocador_comentario podrían escribirse en su lugar como

```
invocador_comentario = YY_START;
```

Flex ofrece YYSTATE como un alias para YY_START (ya que es lo que usa lex de AT&T).

Fíjese que las condiciones de arranque no tienen su propio espacio de nombres; los %s's y %x's declaran nombres de la misma manera que con #define's.

Finalmente, aquí hay un ejemplo de cómo emparejar cadenas entre comillas al estilo de C usando condiciones de arranque exclusivas, incluyendo secuencias de escape expandidas (pero sin incluir la comprobación de cadenas que son demasiado largas):

```
%x str
```

```
%%
```

```
char string_buf[MAX_STR_CONST];  
char *string_buf_ptr;
```

```
\"  string_buf_ptr = string_buf; BEGIN(str);
```

```
<str>\\"  { /* se vio la comilla que cierra - todo está hecho */  
BEGIN(INITIAL);  
*string_buf_ptr = '\0';  
/* devuelve un tipo de token de cadena constante y  
* el valor para el analizador sintáctico  
*/  
}
```

```
<str>\n      {  
    /* error - cadena constante sin finalizar */  
    /* genera un mensaje de error */  
}  
  
<str>\\[0-7]{1,3} {  
    /* secuencia de escape en octal */  
    int resultado;  
  
    (void) sscanf( yytext + 1, "%o", &resultado );  
  
    if ( resultado > 0xff )  
        /* error, constante fuera de rango */  
  
        *string_buf_ptr++ = resultado;  
    }  
  
<str>\\[0-9]+ {  
    /* genera un error - secuencia de escape errónea;  
     * algo como '\48' o '\0777777'  
     */  
}  
  
<str>\\n *string_buf_ptr++ = '\\n';  
<str>\\t *string_buf_ptr++ = '\\t';  
<str>\\r *string_buf_ptr++ = '\\r';  
<str>\\b *string_buf_ptr++ = '\\b';  
<str>\\f *string_buf_ptr++ = '\\f';  
  
<str>\\(.\\n) *string_buf_ptr++ = yytext[1];  
  
<str>[^\\n"]+ {  
    char *yptr = yytext;  
  
    while ( *yprt )  
        *string_buf_ptr++ = *yprt++;  
    }  
}
```

A menudo, como en alguno de los ejemplos anteriores, uno acaba escribiendo un buen número de reglas todas precedidas por la(s) misma(s) condición(es) de arranque. Flex hace esto un poco más fácil y claro introduciendo la noción de ámbito de la condición de arranque. Un ámbito de condición de arranque comienza con:

<SCs>{

Donde SCs es una lista de una o más condiciones de arranque. Dentro del ámbito de la condición de arranque, cada regla automáticamente tiene el prefijo <SCs> aplicado a esta, hasta

un '}' que corresponda con el '{' inicial. Así, por ejemplo,

```
<ESC>{  
    "\n"  return '\n';  
    "\r"  return '\r';  
    "\f"  return '\f';  
    "\0"  return '\0';  
}
```

es equivalente a:

```
<ESC>"\n"  return '\n';  
<ESC>"\r"  return '\r';  
<ESC>"\f"  return '\f';  
<ESC>"\0"  return '\0';
```

Los ámbitos de las condiciones de arranque pueden anidarse.

Están disponibles tres rutinas para manipular pilas de condiciones de arranque:

void yy_push_state(int new_state)

empuja la condición de arranque actual al tope de la pila de las condiciones de arranque y cambia a new_state como si hubiera utilizado BEGIN new_state (recuerde que los nombres de las condiciones de arranque también son enteros).

void yy_pop_state()

extrae el tope de la pila y cambia a este mediante un BEGIN.

int yy_top_state()

devuelve el tope de la pila sin alterar el contenido de la pila.

La pila de las condiciones de arranque crece dinámicamente y por ello no tiene asociada ninguna limitación de tamaño. Si la memoria se agota, se aborta la ejecución del programa.

Para usar pilas de condiciones de arranque, su analizador debe incluir una directiva % option stack (ver Opciones más abajo).

MÚLTIPLES BUFFERS DE ENTRADA

Algunos analizadores (tales como aquellos que aceptan ficheros "incluidos") requieren la lectura de varios flujos de entrada. Ya que los analizadores de flex hacen mucho uso de buffers, uno no puede controlar de dónde será leída la siguiente entrada escribiendo simplemente un YY_INPUT que sea sensible al contexto del análisis.

A YY_INPUT sólo se le llama cuando el analizador alcanza el final de su buffer, que podría ser bastante tiempo después de haber analizado una sentencia como un "include" que requiere el cambio de la fuente de entrada.

Para solventar este tipo de problemas, flex provee un mecanismo para crear y conmutar entre varios buffers de entrada. Un buffer de entrada se crea usando:

YY_BUFFER_STATE yy_create_buffer(FILE *file, int size)

que toma un puntero a FILE y un tamaño "size" y crea un buffer asociado con el fichero dado y lo suficientemente grande para mantener size caracteres (cuando dude, use YY_BUF_SIZE para el tamaño). Este devuelve un handle YY_BUFFER_STATE, que podría pasarse a otras rutinas (ver más abajo). El tipo de YY_BUFFER_STATE es un puntero a una estructura opaca struct yy_buffer_state, de manera que podría inicializar de forma segura variables YY_BUFFER_STATE a ((YY_BUFFER_STATE) 0) si lo desea, y también hacer referencia a la estructura opaca para declarar correctamente buffers de entrada en otros ficheros fuente además de los de su analizador. Fíjese que el puntero a FILE en la llamada a yy_create_buffer se usa solamente como el valor de yyin visto por YY_INPUT; si usted redefine YY_INPUT de manera que no use más a yyin, entonces puede pasar de forma segura un puntero FILE nulo a yy_create_buffer. Se selecciona un buffer en particular a analizar utilizando:

void yy_switch_to_buffer(YY_BUFFER_STATE nuevo_buffer)

conmuta el buffer de entrada del analizador de manera que los tokens posteriores provienen de nuevo_buffer.

Fíjese que yy_switch_to_buffer() podría usarlo yywrap() para arreglar las cosas para un análisis continuo, en lugar de abrir un nuevo fichero y que yyin apunte a este. Fíjese también que cambiar las fuentes de entrada ya sea por medio de yy_switch_to_buffer() o de yywrap() no cambia la condición de arranque.

void yy_delete_buffer(YY_BUFFER_STATE buffer)

se usa para recuperar el almacenamiento asociado a un buffer. (El buffer puede ser nulo, en cuyo caso la rutina no hace nada.) Puede también limpiar el contenido actual de un buffer usando:

void yy_flush_buffer(YY_BUFFER_STATE buffer)

Esta función descarta el contenido del buffer, de manera que la próxima vez que el analizador intente emparejar un token desde el buffer, este primero rellenará el buffer utilizando YY_INPUT.

yy_new_buffer() es un alias de yy_create_buffer(), que se ofrece por compatibilidad con el uso en C++ de new y delete para crear y destruir objetos dinámicos.

Finalmente, la macro YY_CURRENT_BUFFER retorna un handle YY_BUFFER_STATE al buffer actual.

Aquí hay un ejemplo del uso de estas propiedades para escribir un analizador que expande ficheros incluidos (la

propiedad <<EOF>> se comenta más abajo):

```
/* el estado "incl" se utiliza para obtener el nombre
 * del fichero a incluir.
 */
%<<x incl
```

```
%{  
#define MAX_INCLUDE_DEPTH 10  
YY_BUFFER_STATE include_stack[MAX_INCLUDE_DEPTH];  
int include_stack_ptr = 0;  
%}  
  
%%  
include      BEGIN(incl);  
  
[a-z]+      ECHO;  
[^a-zA-Z]*\n?  ECHO;  
  
<incl>[\t]* /* se come los espacios en blanco */  
<incl>[^ \t\n]+ { /* obtiene el nombre de fichero a incluir */  
if ( include_stack_ptr >= MAX_INCLUDE_DEPTH )  
{  
    fprintf( stderr, "Demasiados include anidados" );  
    exit( 1 );  
}  
  
include_stack[include_stack_ptr++] =  
YY_CURRENT_BUFFER;  
  
yyin = fopen( yytext, "r" );  
  
if ( ! yyin )  
    error( ... );  
  
yy_switch_to_buffer(  
    yy_create_buffer( yyin, YY_BUF_SIZE ));  
  
BEGIN(INITIAL);  
}  
  
<<EOF>> {  
if ( --include_stack_ptr < 0 )  
{  
    yyterminate();  
}  
  
else  
{  
    yy_delete_buffer( YY_CURRENT_BUFFER );  
    yy_switch_to_buffer(  
        include_stack[include_stack_ptr] );  
}  
}
```

Se dispone de tres rutinas para preparar buffers de entrada para el análisis de cadenas en memoria en lugar de archivos. Todas estas crean un nuevo buffer de entrada para analizar la cadena, y devuelven el correspondiente handle YY_BUFFER_STATE (que usted debería borrar con yy_delete_buffer() cuando termine con él). Estas también comután el nuevo buffer usando yy_switch_to_buffer(), de manera que la próxima llamada a yylex() comenzará analizando la cadena.

yy_scan_string(const char *str)
analiza una cadena terminada en nulo.

yy_scan_bytes(const char *bytes, int len)
analiza len bytes (incluyendo posibles NUL's) comenzando desde el punto bytes.

Fíjese que ambas de estas funciones crean y analizan una copia de la cadena o bytes. (Esto podría ser deseable, ya que yylex() modifica el contenido del buffer que está analizado.) Usted puede evitar la copia utilizando:

yy_scan_buffer(char *base, yy_size_t size)

que analiza in situ el buffer comenzando en base, que consiste de size bytes, donde los dos últimos bytes deben ser YY_END_OF_BUFFER_CHAR (ASCII NUL). Estos dos últimos bytes no se analizan; así, el análisis consta de base[0] hasta base[size-2], inclusive.

Si se equivoca al disponer base de esta manera (es decir, olvidar los dos YY_END_OF_BUFFER_CHAR bytes finales), entonces yy_scan_buffer() devuelve un puntero nulo en lugar de crear un nuevo buffer de entrada.

El tipo yy_size_t es un tipo entero con el que puede hacer una conversión a una expresión entera para reflejar el tamaño del buffer.

REGLAS DE FIN-DE-FICHERO

La regla especial "<<EOF>>" indica las acciones que deben tomarse cuando se encuentre un fin-de-fichero e yywrap() retorne un valor distinto de cero (es decir, indica que no quedan ficheros por procesar). La acción debe finalizar haciendo una de estas cuatro cosas:

- asignando a yyin un nuevo fichero de entrada (en versiones anteriores de flex, después de hacer la asignación debía llamar a la acción especial YY_NEW_FILE; esto ya no es necesario);
- ejecutando una sentencia return;
- ejecutando la acción especial yyterminate();
- o, comutando a un nuevo buffer usando yy_switch_to_buffer() como se mostró en el ejemplo anterior.

Las reglas <<EOF>> no deberían usarse con otros patrones; estas deberían calificarse con una lista de condiciones de arranque. Si se da una regla <<EOF>> sin calificar, esta se aplica a todas las condiciones de arranque que no tengan ya acciones <<EOF>>. Para especificar una regla <<EOF>> solamente para la condición de arranque inicial, use

<INITIAL><<EOF>>

Estas reglas son útiles para atrapar cosas tales como comentarios sin final.

Un ejemplo:

```
%x comilla
%%
```

...otras reglas que tengan que ver con comillas...

```
<comilla><<EOF>> {
    error( "comilla sin cerrar" );
    yyterminate();
}
<<EOF>> {
    if ( *++filelist )
        yyin = fopen( *filelist, "r" );
    else
        yyterminate();
}
```

MACROS MISCELÁNEAS

La macro YY_USER_ACTION puede definirse para indicar una acción que siempre se ejecuta antes de la acción de la regla emparejada. Por ejemplo, podría declararse con #define para que llame a una rutina que convierta yytext a minúsculas. Cuando se invoca a YY_USER_ACTION, la variable yy_act da el número de la regla emparejada (las reglas están numeradas comenzando en 1). Suponga que quiere medir la frecuencia con la que sus reglas son emparejadas. Lo que viene a continuación podría hacer este truco:

```
#define YY_USER_ACTION ++ctr[yy_act]
```

donde ctr es un vector que mantiene la cuenta para las diferentes reglas. Fíjese que la macro YY_NUM_RULES da el número total de reglas (incluyendo la regla por defecto, incluso si usted usa -s), así que una declaración correcta para ctr es:

```
int ctr[YY_NUM_RULES];
```

La macro YY_USER_INIT podría definirse para indicar una acción que siempre se ejecuta antes del primer análisis (y antes de que se haga la inicialización interna del analizador). Por ejemplo, este podría usarse para llamar a una rutina que lea una tabla de datos o abrir un fichero de registro.

La macro yy_set_interactive(is_interactive) se puede usar para controlar si el buffer actual se considera interactivo. Un buffer interactivo se procesa más lentamente, pero debe usarse cuando la fuente de entrada del analizador es realmente interactiva para evitar problemas debidos a la espera para el llenado de los buffers (ver el comentario de la bandera -I más abajo). Un valor distinto de cero en la invocación de la macro marcará el buffer como interactivo, un valor de cero como no-interactivo. Fíjese que el uso de esta macro no tiene en cuenta %option always-interactive o %option never-interactive (ver Opciones más abajo). yy_set_interactive() debe invocarse antes del comienzo del análisis del buffer que es considerado (o no) interactivo.

La macro yy_set_bol(at_bol) puede usarse para controlar si el contexto del buffer de análisis actual para el próximo emparejamiento de token se hace como si se encontrara al principio de

una línea. Un argumento de la macro distinto de cero hace activas a las reglas sujetas a '^', mientras que un argumento igual a cero hace inactivas a las reglas con '^'.

La macro YY_AT_BOL() devuelve verdadero si el próximo token analizado a partir del buffer actual tendrá activas las reglas '^', de otra manera falso.

En el analizador generado, las acciones están recogidas en una gran sentencia switch y separadas usando YY_BREAK, que puede ser redefinida. Por defecto, este es simplemente un "break", para separar la acción de cada regla de las reglas que le siguen. Redefiniendo YY_BREAK permite, por ejemplo, a los usuarios de C++ que #define YY_BREAK no haga nada (¡mientras tengan cuidado para que cada regla finalice con un "break" o un "return"!) para evitar que sufran los avisos de sentencias inalcanzables cuando debido a que la acción de la regla finaliza con un "return", el YY_BREAK es inaccesible.

VALORES DISPONIBLES AL USUARIO

Esta sección resume los diferentes valores disponibles al usuario en las acciones de la regla.

- **char *yytext apunta al texto del token actual. Este puede modificarse pero no alargarse (no puede añadir caracteres al final).**

Si aparece la directiva especial %array en la primera sección de la descripción del analizador, entonces yytext se declara en su lugar como char yytext[YYLMAX], donde YYLMAX es la definición de una macro que puede redefinir en la primera sección si no le gusta el valor por defecto (generalmente 8KB). El uso de %array produce analizadores algo más lentos, pero el valor de yytext se vuelve inmune a las llamadas a input() y unput(), que potencialmente destruyen su valor cuando yytext es un puntero a carácter. El opuesto de %array es %pointer, que se encuentra por defecto.

Usted no puede utilizar %array cuando genera analizadores como clases de C++ (la bandera -+).

- **int yyleng contiene la longitud del token actual.**
- **FILE *yyin es el fichero por el que flex lee por defecto. Este podría redefinirse pero hacerlo solo tiene sentido antes de que el análisis comience o después de que se haya encontrado un EOF. Cambiándolo en medio del análisis tendrá resultados inesperados ya que flex utiliza buffers en su entrada; use yyrestart() en su lugar. Una vez que el análisis termina debido a que se ha visto un fin-de-fichero, puede asignarle a yyin el nuevo fichero de entrada y entonces llamar al analizador de nuevo para continuar analizando.**
- **void yyrestart(FILE *new_file) podría ser llamada para que yyin apunte al nuevo fichero de entrada. El cambio al nuevo fichero es inmediato (cualquier entrada contenida en el buffer previamente se pierde).**

Fíjese que llamando a yyrestart() con yyin como argumento de esta manera elimina el buffer de entradda actual y continúa analizando el mismo fichero de entrada.

- **FILE *yyout es el fichero sobre el que se hacen las acciones ECHO. Este puede ser reasignado por el usuario.**
- **YY_CURRENT_BUFFER devuelve un handle YY_BUFFER_STATE al buffer actual.**
- **YY_START devuelve un valor entero correspondiente a la condición de arranque actual. Posteriormente puede usar este valor con BEGIN para retornar a la condición de arranque.**

INTERFAZ CON YACC

Uno de los usos principales de flex es como compañero del generador de analizadores sintácticos yacc. Los analizadores de yacc esperan invocar a una rutina llamada `yylex()` para encontrar el próximo token de entrada. La rutina se supone que devuelve el tipo del próximo token además de poner cualquier valor asociado en la variable global `yyval`. Para usar flex con yacc, uno especifica la opción `-d` de yacc para intruirle a que genere el fichero `y.tab.h` que contiene las definiciones de todos los %tokens que aparecen en la entrada de yacc. Entonces este archivo se incluye en el analizador de flex. Por ejemplo, si uno de los tokens es "TOK_NUMERO", parte del analizador podría parecerse a:

```
%{  
#include "y.tab.h"  
%}  
  
% %  
  
[0-9]+ yyval = atoi( yytext ); return TOK_NUMERO;
```

OPCIONES

flex tiene las siguientes opciones:

-b Genera información de retroceso en `lex.backup`. Esta es una lista de estados del analizador que requieren retroceso y los caracteres de entrada con los que la hace. Añadiendo reglas uno puede eliminar estados de retroceso. Si todos los estados de retroceso se eliminan y se usa `-Cf` o `-CF`, el analizador generado funcionará más rápido (ver la bandera `-p`). Únicamente los usuarios que desean exprimir hasta el último ciclo de sus analizadores necesitan preocuparse de esta opción. (Ver la sección sobre Consideraciones de Rendimiento más abajo.)

-c es una opción que no hace nada, incluída para cumplir con POSIX.

-d hace que el analizador generado se ejecute en modo de depuración. Siempre que se reconoce un patrón y la variable global `yy_flex_debug` no es cero (que por defecto no lo es), el analizador escribirá en `stderr` una línea de la forma:

--accepting rule at line 53 ("el texto emparejado")

El número de línea hace referencia al lugar de la regla en el fichero que define al analizador (es decir, el fichero que se le introdujo a flex). Los mensajes también se generan cuando el analizador retrocede, acepta la regla por defecto, alcanza el final de su buffer de entrada (o encuentra un NUL; en este punto, los dos parecen lo mismo en lo que le concierne al analizador), o alcance el fin-de-fichero.

-f especifica un analizador rápido. No se realiza una compresión de tablas y se evita el uso de `stdio`. El resultado es grande pero rápido. Esta opción es equivalente a `-Cfr` (ver más abajo).

-h genera un sumario de "ayuda" de las opciones de flex por `stdout` y entonces finaliza. **-?** y **--help** son sinónimos de **-h**.

-i indica a flex que genere un analizador case-insensitive. Se ignorará si las letras en los patrones de entrada de flex son en mayúsculas o en minúsculas, y los tokens en la entrada serán emparejados sin tenerlo en cuenta. El texto emparejado dado en yytext tendrá las mayúsculas y minúsculas preservadas (es decir, no se convertirán).

-l activa el modo de máxima compatibilidad con la implementación original de lex de AT&T. Fíjese que esto no significa una compatibilidad completa. El uso de esta opción cuesta una cantidad considerable de rendimiento, y no puede usarse con las opciones **-+**, **-f**, **-F**, **-Cf**, o **-CF**. Para los detalles a cerca de la compatibilidad que se ofrece, vea la sección "Incompatibilidades con Lex y POSIX" más abajo. Esta opción también hace que se defina el nombre YY_FLEX_LEX_COMPAT en el analizador generado.

-n es otra opción que no hace nada, incluída para cumplir con POSIX.

-p genera un informe de rendimiento en stderr. El informe consta de comentarios que tratan de las propiedades del fichero de entrada de flex que provocarán pérdidas serias de rendimiento en el analizador resultante. Si indica esta bandera dos veces, también obtendrá comentarios que tratan de las propiedades que producen pérdidas menores de rendimiento.

Fíjese que el uso de REJECT, %option yylineno, y el contexto posterior variable (vea la sección Deficiencias / Errores más abajo) supone una penalización substancial del rendimiento; el uso de yymore(), el operador ^, y la bandera -I supone penalizaciones del rendimiento menores.

-s hace que la regla por defecto (que la entrada sin emparejar del analizador se repita por stdout) se suprima. Si el analizador encuentra entrada que no es reconocida por ninguna de sus reglas, este aborta con un error. Esta opción es útil para encontrar agujeros en el conjunto de reglas del analizador.

-t indica a flex que escriba el analizador que genera a la salida estándar en lugar de en lex.yy.c.

-v especifica que flex debería escribir en stderr un sumario de estadísticas respecto al analizador que genera. La mayoría de las estadísticas no tienen significado para el usuario casual de flex, pero la primera línea identifica la versión de flex (la misma que se informa con **-V**), y la próxima línea las banderas utilizadas cuando se genera el analizador, incluyendo aquellas que se encuentran activadas por defecto.

-w suprime los mensajes de aviso.

-B dice a flex que genere un analizador batch, que es lo opuesto al analizador interactivo generador por **-I** (ver más abajo). En general, use **-B** cuando esté seguro de que su analizador nunca se usará de forma interactiva, y quiere con esto exprimir un poco más el rendimiento. Si por el contrario su objetivo es exprimirlo mucho más, debería estar utilizando la opción **-Cf** o **-CF** (comentadas más abajo), que activa **-B** automáticamente de todas maneras.

-F especifica que se debe utilizar la representación de la tabla rápida (y elimina referencias a stdio). Esta representación es aproximadamente tan rápida como la representación completa de la tabla (-f), y para algunos conjuntos de patrones será considerablemente más pequeña (y para otros, mayor). En general, si el conjunto de patrones contiene "palabras clave" y una regla "identificador" atrápalo-todo, como la del conjunto:

```
"case"  return TOK_CASE;
"switch" return TOK_SWITCH;
...
"default" return TOK_DEFAULT;
[a-z]+  return TOK_ID;
```

entonces será mejor que utilice la representación de la tabla completa. Si sólo está presente la regla "identificador" y utiliza una tabla hash o algo parecido para detectar palabras clave, mejor utilice -F. Esta opción es equivalente a -Cf (ver más abajo). Esta opción no puede utilizarse con -+.

-I ordena a flex que genere un analizador interactivo. Un analizador interactivo es uno que solo mira hacia delante para decidir que token ha sido reconocido únicamente si debe hacerlo. Resulta que mirando siempre un carácter extra hacia delante, incluso si el analizador ya ha visto suficiente texto para eliminar la ambigüedad del token actual, se es un poco más rápido que mirando solamente cuando es necesario. Pero los analizadores que siempre miran hacia delante producen un comportamiento interactivo malísimo; por ejemplo, cuando un usuario teclea una línea nueva, esta no se reconoce como un token de línea nueva hasta que introduzca otro token, que a menudo significa introducir otra línea completa.

Los analizadores de flex por defecto son interactivos a menos que use la opción -Cf o -CF de compresión de tablas (ver más abajo). Esto es debido a que si está buscando un rendimiento alto tendría que estar utilizando una de estas opciones, así que si no lo ha hecho flex asume que prefiere cambiar un poco de rendimiento en tiempo de ejecución en beneficio de un comportamiento interactivo intuitivo. Fíjese también que no puede utilizar -I conjuntamente con -Cf o -CF. Así, esta opción no se necesita realmente; está activa por defecto para todos esos casos en los que se permite.

Usted puede forzar al analizador que no sea interactivo usando -B (ver más arriba).

-L ordena a flex que no genere directivas #line. Sin esta opción, flex acarilla al analizador generado con directivas #line para que los mensajes de error en las acciones estén localizadas correctamente respecto al fichero original de flex (si los errores son debidos al código en el fichero de entrada), o a lex.yy.c (si los errores son fallos de flex -- debería informar de este tipo de errores a la dirección de correo dada más abajo).

-T hace que flex se ejecute en modo de traza. Este generará un montón de mensajes en stderr relativos a la forma de la entrada y el autómata finito no-determinista o determinista resultante. Esta opción generalmente es para usarla en el mantenimiento de flex.

-V imprime el número de la versión en stdout y sale. --version es un sinónimo de -V.

-7 ordena a flex que genere un analizador de 7-bits, es decir, uno que sólo puede reconocer

caracteres de 7-bits en su entrada. La ventaja de usar -7 es que las tablas del analizador pueden ser hasta la mitad del tamaño de aquellas generadas usando la opción -8 (ver más abajo). La desventaja es que tales analizadores a menudo se cuelgan o revientan si su entrada contiene caracteres de 8-bits.

Fíjese, sin embargo, que a menos que genere su analizador utilizando las opciones de compresión de tablas -Cf o -CF, el uso de -7 ahorrará solamente una pequeña cantidad de espacio en la tabla, y hará su analizador considerablemente menos portable. El comportamiento por defecto de flex es generar un analizador de 8-bits a menos que use -Cf o -CF, en cuyo caso flex por defecto genera analizadores de 7-bits a menos que su sistema siempre esté configurado para generar analizadores de 8-bits (a menudo este será el caso de los sistemas fuera de EEUU). Puede decir si flex generó un analizador de 7 u 8 bits inspeccionando el sumario de banderas en la salida de -v como se describió anteriormente.

Fíjese que si usa -Cfe o -CFe (esas opciones de compresión de tablas, pero también el uso de clases de equivalencia como se comentará más abajo), flex genera aún por defecto un analizador de 8-bits, ya que normalmente con estas opciones de compresión las tablas de 8-bits completas no son mucho más caras que las tablas de 7-bits.

-8 ordena a flex que genere un analizador de 8-bits, es decir, uno que puede reconocer caracteres de 8-bits.

Esta bandera sólo es necesaria para analizadores generados usando -Cf o -CF, ya que de otra manera flex por defecto genera un analizador de 8-bits de todas formas.

Vea el comentario sobre -7 más arriba a cerca del comportamiento por defecto de flex y la discusión entre los analizadores de 7-bits y 8-bits.

-+ especifica que quiere que flex genere un analizador como una clase de C++. Vea la sección Generando Escáneres en C++ más abajo para los detalles.

-C[aefFmr]

controla el grado de compresión de la tabla y, más generalmente, el compromiso entre analizadores pequeños y analizadores rápidos.

-Ca ("alinea") ordena a flex que negocie tablas más grandes en el analizador generado para un comportamiento más rápido porque los elementos de las tablas están mejor alineados para el acceso a memoria y computación. En algunas arquitecturas RISC, la búsqueda y manipulación de palabras largas es más eficiente que con unidades más pequeñas tales como palabras cortas. Esta opción puede doblar el tamaño de las tablas usadas en su analizador.

-Ce ordena a flex que construya clases de equivalencia, es decir, conjunto de caracteres que tienen identicas propiedades léxicas (por ejemplo, si la única aparición de dígitos en la entrada de flex es en la clase de caracteres "[0-9]" entonces los dígitos '0', '1', ..., '9' se pondrán todos en la misma clase de equivalencia). Las clases de equivalencia normalmente ofrecen notables reducciones en los tamaños de los ficheros finales de tabla/objeto (típicamente un factor de 2-5) y son juiciosamente bastante baratos en cuanto al rendimiento (una localización en un vector por carácter analizado).

-Cf especifica que se deben generar las tablas del analizador completas - flex no debería comprimir las tablas tomando ventaja de las funciones de transición similares para diferentes estados.

-CF especifica que debería usarse la representación del analizador rápido alternativo (descrito anteriormente en la bandera -F) Esta opción no puede usarse con -+.

-Cm ordena a flex a que construya clases de meta-equivalencias, que son conjuntos de clases de equivalencia (o caracteres, si las clases de equivalencia no se están usando) que comúnmente se usan de forma conjunta. Las clases de meta-equivalencias son a menudo un gran ahorro cuando se usan tablas comprimidas, pero tienen un impacto moderado en el rendimiento (uno o dos tests "if" y una localización en un array por carácter analizado).

-Cr hace que el analizador generado elimine el uso de la librería de E/S estándar para la entrada. En lugar de llamar a fread() o getc(), el analizador utilizará la llamada al sistema read(), produciendo una ganancia en el rendimiento que varía de sistema en sistema, pero en general probablemente es insignificante a menos que también esté usando -Cf o -CF. El uso de -Cr puede producir un comportamiento extraño si, por ejemplo, lee de yyin usando stdio antes de llamar al analizador (porque el analizador perderá cualquier texto que sus lecturas anteriores dejaron en el buffer de entrada de stdio).

-Cr no tiene efecto si usted define YY_INPUT (ver El Escáner Generado más arriba).

Con solamente -C se especifica que las tablas del analizador deberían comprimirse pero no debería utilizarse ni las clases de equivalencia ni las clases de meta-equivalencias.

Las opciones -Cf o -CF y -Cm no tienen sentido juntas - no hay oportunidad para las clases de meta-equivalencias si la tabla no está siendo comprimida. De otra forma las opciones podrían mezclarse libremente, y son acumulativas.

La configuración por defecto es -Cem, que especifica que flex debería generar clases de equivalencia y clases de meta-equivalencias. Esta configuración provee el mayor grado de compresión. Puede llegarse a un compromiso entre analizadores de ejecución más rápida con el coste de tablas mayores siendo generalmente verdadero lo siguiente:

lo más lento y pequeño

- Cem**
- Cm**
- Ce**
- C**
- C{f,F}e**
- C{f,F}**
- C{f,F}a**

lo más rápido y grande

Fíjese que los analizadores con tablas más pequeñas normalmente se generan y compilan de la forma más rápida posible, así que durante el desarrollo usted normalmente querrá usar como viene por defecto, compresión máxima.

-Cfe a menudo es un buen compromiso entre velocidad y tamaño para la producción de analizadores.

-o salida

ordena a flex que escriba el analizador al fichero salida en lugar de a lex.yy.c. Si combina -o con la opción -t, entonces el analizador se escribe en stdout pero sus directivas #line (vea la opción -L más arriba) hacen referencia al fichero salida.

-P prefijo

cambia el prefijo yy usado por defecto por flex para todas las variables visibles globalmente y nombres de funciones para que sea prefijo. Por ejemplo, -Pfoo cambia el nombre de yytext a footext. Este también cambia el nombre por defecto del fichero de salida de lex.yy.c a lex.foo.c. Aquí están todos los nombres afectados:

```
yy_create_buffer
yy_delete_buffer
yy_flex_debug
yy_init_buffer
yy_flush_buffer
yy_load_buffer_state
yy_switch_to_buffer
yyin
yyleng
yylex
yylineno
yyout
yyrestart
yytext
yywrap
```

(Si usted está utilizando un analizador en C++, entonces únicamente yywrap y yyFlexLexer se ven afectados.) Dentro de su analizador, puede aún hacer referencia a las variables globales y funciones usando cualquier versión de su nombre; pero externamente, estas tienen el nombre modificado.

Esta opción le deja enlazar fácilmente múltiples programas flex conjuntamente en el mismo ejecutable.

Fíjese, sin embargo, que usando esta opción también se renombra yywrap(), de manera que ahora debe o bien proveer su propia versión de la rutina (con el nombre apropiado) para su analizador, o usar %option noyywrap, ya que enlazar con -lfl no podrá proveerle una por defecto.

-Sfichero_esqueleto

ignora el fichero de esqueleto por defecto con el que flex construye sus analizadores. Usted probablemente nunca necesitará utilizar esta opción a menos que este haciendo mantenimiento o un desarrollo de flex.

flex también ofrece un mecanismo para controlar las opciones dentro de la propia especificación del analizador, en vez de a partir de la línea de comando. Esto se hace incluyendo

las directivas %option en la primera sección de la especificación del analizador. Usted puede especificar varias opciones con una sola directiva %option, y varias directivas en la primera sección de su fichero de entrada de flex.

La mayoría de las opciones vienen dadas simplemente como nombres, opcionalmente precedidos por la palabra "no" (sin intervenir un espacio) para negar su significado. Las banderas de flex o su negación son equivalentes a un número:

7bit	opción -7
8bit	opción -8
align	opción -Ca
backup	opción -b
batch	opción -B
c++	opción +-+

caseful o
case-sensitive opuesto de -i (por defecto)

case-insensitive o
caseless opción -i

debug	opción -d
default	opuesto de la opción -s
ecs	opción -Ce
fast	opción -F
full	opción -f
interactive	opción -I
lex-compat	opción -l
meta-ecs	opción -Cm
perf-report	opción -p
read	opción -Cr
stdout	opción -t
verbose	opción -v
warn	opuesto de la opción -w

(use "%option nowarn" para -w)

array	equivalente a "%array"
pointer	equivalente a "%pointer" (por defecto)

Algunas directivas %option ofrecen propiedades que de otra manera no están disponibles:

always-interactive

ordena a flex que genere un analizador que siempre considere su entrada como "interactiva". Normalmente, sobre cada fichero de entrada nuevo el analizador llama a isatty() como intento para determinar si la entrada del analizador es interactiva y por lo tanto debería leer un carácter a la vez. Cuando esta opción se utilice, sin embargo, entonces no se hace tal llamada.

main ordena a flex que facilite un programa main() por defecto para el analizador, que simplemente llame a yylex(). Esta opción implica noyywrap (ver más abajo).

never-interactive

ordena a flex que genere un analizador que nunca considere su entrada como "interactiva" (de nuevo, no se hace ninguna llamada a isatty()). Esta es la opuesta a always-interactive.

stack activa el uso de pilas de condiciones de arranque (ver Condiciones de Arranque más arriba).

stdinit

si se establece (es decir, %option stdinit) inicializa yyin e yyout a stdin y stdout, en lugar del que viene por defecto que es nil. Algunos programas de lex existentes dependen de este comportamiento, incluso si no sigue el ANSI C, que no requiere que stdin y stdout sean constantes en tiempo de compilación.

yylineno

ordena a flex a generar un analizador que mantenga el número de la línea actual leída desde su entrada en la variable global yylineno. Esta opción viene implícita con %option lex-compat.

yywrap si no se establece (es decir, %option noyywrap), hace que el analizador no llame a yywrap() hasta el fin-de-fichero, pero simplemente asume que no hay más ficheros que analizar (hasta que el usuario haga apuntar yyin a un nuevo fichero y llame a yylex() otra vez).

flex analiza las acciones de sus reglas para determinar si utiliza las propiedades REJECT o yymore(). Las opciones reject e yymore están disponibles para ignorar sus decisiones siempre que use las opciones, o bien estableciéndolas (p.ej., %option reject) para indicar que la propiedad se utiliza realmente, o desactivándolas para indicar que no es utilizada (p.ej., %option noyymore).

Tres opciones toman valores delimitados por cadenas, separadas por '=':

%option outfile="ABC"

es equivalente a -oABC, y

%option prefix="XYZ"

es equivalente a -PXYZ. Finalmente,

%option yyclass="foo"

sólo se aplica cuando se genera un analizador en C++ (opción -+). Este informa a flex que ha derivado a foo como una subclase de yyFlexLexer, así que flex pondrá sus acciones en la función miembro foo::yylex() en lugar de yyFlexLexer::yylex(). Este también genera una función miembro yyFlexLexer::yylex() que emite un error en tiempo de ejecución (invocando a

yyFlexLexer::LexerError() si es llamada. Ver Generando Escáneres en C++, más abajo, para información adicional.

Están disponibles un número de opciones para los puristas de lint que desean suprimir la aparición de rutinas no necesarias en el analizador generado. Cada una de la siguientes, si se desactivan (p.ej., `%option nounput`), hace que la rutina correspondiente no aparezca en el analizador generado:

input, unput
yy_push_state, yy_pop_state, yy_top_state
yy_scan_buffer, yy_scan_bytes, yy_scan_string

(aunque `yy_push_state()` y sus amigas no aparecerán de todas manera a menos que use `%option stack`).

CONSIDERACIONES DE RENDIMIENTO

El principal objetivo de diseño de flex es que genere analizadores de alto rendimiento. Este ha sido optimizado para comportarse bien con conjuntos grandes de reglas. Aparte de los efectos sobre la velocidad del analizador con las opciones de compresión de tablas -C anteriormente introducidas, hay un número de opciones/acciones que degradan el rendimiento. Estas son, desde la más costosa a la menos:

REJECT
%option yylineno
contexto posterior arbitrario

conjunto de patrones que requieren retroceso
%array
%option interactive
%option always-interactive

'^' operador de comienzo de línea
yymore()

siendo las tres primeras bastante costosas y las dos últimas bastante económicas. Fíjese también que `unput()` se implementa como una llamada de rutina que potencialmente hace bastante trabajo, mientras que `yyless()` es una macro bastante económica; así que si está devolviendo algún texto excedente que ha analizado, use `yyless()`.

REJECT debería evitarse a cualquier precio cuando el rendimiento es importante. Esta es una opción particularmente cara.

Es lioso deshacerse del retroceso y a menudo podría ser una cantidad de trabajo enorme para un analizador complicado. En principio, uno comienza utilizando la bandera `-b` para generar un archivo `lex.backup`. Por ejemplo, sobre la entrada

```
%%
foo    return TOK_KEYWORD;
```

```
foobar    return TOK_KEYWORD;
```

el fichero tiene el siguiente aspecto:

El estado #6 es no-aceptar -
números de línea asociados a la regla:
2 3
fin de transiciones: [o]
transiciones de bloqueo: fin de archivo (EOF) [\001-n p-\177]

El estado #8 es no-aceptar -
números de línea asociados a la regla:
3
fin de transiciones: [a]
transiciones de bloqueo: fin de archivo (EOF) [\001-` b-\177]

El estado #9 es no-aceptar -
números de línea asociados a la regla:
3
fin de transiciones: [r]
transiciones de bloqueo: fin de archivo (EOF) [\001-q s-\177]

Las tablas comprimidas siempre implican un retroceso.

Las primeras líneas nos dicen que hay un estado del analizador en el que se puede hacer una transición con una 'o' pero no sobre cualquier otro carácter, y que en ese estado el texto recientemente analizado no empareja con ninguna regla. El estado ocurre cuando se intenta emparejar las reglas encontradas en las líneas 2 y 3 en el fichero de entrada. Si el analizador está en ese estado y entonces lee cualquier cosa que no sea una 'o', tendrá que retroceder para encontrar una regla que empareje. Con un poco de análisis uno puede ver que este debe ser el estado en el que se está cuando se ha visto "fo". Cuando haya ocurrido, si se ve cualquier cosa que no sea una 'o', el analizador tendrá que retroceder para simplemente emparejar la 'f' (por la regla por defecto).

El comentario que tiene que ver con el Estado #8 indica que hay un problema cuando se analiza "foob". En efecto, con cualquier carácter que no sea una 'a', el analizador tendrá que retroceder para aceptar "foo". De forma similar, el comentario para el Estado #9 tiene que ver cuando se ha analizado "fooba" y no le sigue una 'r'.

El comentario final nos recuerda que no merece la pena todo el trabajo para eliminar el retroceso de las reglas a menos que estemos usando -Cf o -CF, y que no hay ninguna mejora del rendimiento haciéndolo con analizadores comprimidos.

La manera de quitar los retrocesos es añadiendo reglas de "error":

```
% %  
foo    return TOK_KEYWORD;  
foobar  return TOK_KEYWORD;  
  
fooba  |
```

```

foob    |
fo      {
    /* falsa alarma, realmente no es una palabra clave */
    return TOK_ID;
}

```

La eliminación de retroceso en una lista de palabras clave también puede hacerse utilizando una regla "atrápalo-todo":

```

% %
foo    return TOK_KEYWORD;
foobar return TOK_KEYWORD;

[a-z]+  return TOK_ID;

```

Normalmente esta es la mejor solución cuando sea adecuada.

Los mensajes sobre retrocesos tienden a aparecer en cascada. Con un conjunto complicado de reglas no es poco común obtener cientos de mensajes. Si uno puede descifrarlos, sin embargo, a menudo sólo hay que tomar una docena de reglas o algo así para eliminar los retrocesos (ya que es fácil cometer una equivocación y tener una regla de error que reconozca un token válido. Una posible característica futura de flex será añadir reglas automáticamente para eliminar el retroceso).

Es importante tener en cuenta que se obtienen los beneficios de eliminar el retroceso sólo si elimina cada instancia del retroceso. Dejar solamente una significa que no ha ganado absolutamente nada.

El contexto posterior variable (donde la parte delantera y posterior no tienen una longitud fija) supone casi la misma pérdida de rendimiento que REJECT (es decir, substanciales). Así que cuando sea posible una regla como esta:

```

% %
raton|rata/(gato|perro)  correr();

```

es mejor escribirla así:

```

% %
raton/gato|perro      correr();
rata/gato|perro      correr();

```

o así

```

% %
raton|rata/gato      correr();
raton|rata/perro     correr();

```

Fíjese que aquí la acción especial '|' no ofrece ningún ahorro, y puede incluso hacer las cosas peor (ver Deficiencias / Errores más abajo).

Otro área donde el usuario puede incrementar el rendimiento del analizador (y una que es más fácil de implementar) surge del hecho que cuanto más tarde se empareje un token, más rápido irá el analizador. Esto es debido a que con tokens grandes el procesamiento de la mayoría de los caracteres de entrada tiene lugar en el (corto) bucle de análisis más interno, y no tiene que ir tan a menudo a hacer el trabajo de más para constituir el entorno del analizador (p.ej., yytext) para la acción. Recuerde el analizador para los comentarios en C:

```
%x comentario
%
int num_linea = 1;

"/*"      BEGIN(comentario);

<comentario>[^*\n]*
<comentario>"*"+[^*\n]*
<comentario>\n      ++num_linea;
<comentario>"*"/"      BEGIN(INITIAL);
```

Esto podría acelerarse escribiéndolo como:

```
%x comentario
%
int num_linea = 1;

"/*"      BEGIN(comentario);

<comentario>[^*\n]*
<comentario>[^*\n]*\n      ++num_linea;
<comentario>"*"+[^*\n]*
<comentario>"*"+[^*\n]*\n      ++num_linea;
<comentario>"*"/"      BEGIN(INITIAL);
```

Ahora en lugar de que cada línea nueva requiera el procesamiento de otra regla, el reconocimiento de las líneas nuevas se "distribuye" sobre las otras reglas para mantener el texto reconocido tan largo como sea posible.

Fíjese que el añadir reglas no ralentiza el analizador! La velocidad del analizador es independiente del número de reglas o (dadas las consideraciones dadas al inicio de esta sección) cuán complicadas sean las reglas respecto a operadores tales como '*' y '|'.

Un ejemplo final sobre la aceleración de un analizador: suponga que quiere analizar un fichero que contiene identificadores y palabras clave, una por línea y sin ningún carácter extraño, y reconocer todas las palabras clave. Una primera aproximación natural es:

```
%
asm  |
auto |
break |
... etc ...
```

```

volatile |
while /* es una palabra clave */

.\n /* no es una palabra clave */

```

Para eliminar el retroceso, introduzca una regla atrápalo-todo:

```

% %
asm |
auto |
break |
... etc ...
volatile |
while /* es una palabra clave */

```

```

[a-z]+ |
.\n /* no es una palabra clave */

```

Ahora, si se garantiza que hay ex  ctamente una palabra por l  nea, entonces podemos reducir el n  mero total de emparejamientos por la mitad mezclando el reconocimiento de l  neas nuevas con las de los otros tokens:

```

% %
asm\n |
auto\n |
break\n |
... etc ...
volatile\n |
while\n /* es una palabra clave */

```

```

[a-z]+\n |
.\n /* no es una palabra clave */

```

Uno tiene que ser cuidadoso aqu , ya que hemos reintroducido retroceso en el analizador. En particular, aunque nosotros sepamos que ah  nunca habr n otros caracteres en el flujo de entrada que no sean letras o l  neas nuevas, flex no puede figurarse eso, y planear  la posible necesidad de retroceder cuando haya analizado un token como "auto" y el pr ximo caracter sea algo distinto a una l  nea nueva o una letra. Previamente este podr a entonces emparejar la regla "auto" y estar todo hecho, pero ahora este no tiene una regla "auto", solamente una regla "auto\n". Para eliminar la posibilidad de retroceso, podr amos o bien duplicar todas las reglas pero sin l  nea nueva al final, o, ya que nunca esperamos encontrar tal entrada y por lo tanto ni c mo es clasificada, podemos introducir una regla atr palo-todo m s, esta que no incluye una l  nea nueva:

```

% %
asm\n |
auto\n |

```

```
break\n |
... etc ...
volatile\n |
while\n /* es una palabra clave */

[a-z]+\n |
[a-z]+ |
.\n /* no es una palabra clave */
```

Compilado con -Cf, esto es casi tan rápido como lo que uno puede obtener de un analizador de flex para este problema en particular.

Una nota final: flex es lento cuando empareja NUL's, particularmente cuando un token contiene múltiples NUL's.

Es mejor escribir reglas que emparejen cortas cantidades de texto si se anticipa que el texto incluirá NUL's a menudo.

Otra nota final en relación con el rendimiento: tal y como se mencionó en la sección Cómo se Reconoce la Entrada, el reajuste dinámico de yytext para acomodar tokens enormes es un proceso lento porque ahora requiere que el token (inmenso) sea reanalizado desde el principio. De esta manera si el rendimiento es vital, debería intentar emparejar "grandes" cantidades de texto pero no "inmensas" cantidades, donde el punto medio está en torno a los 8K caracteres/token.

GENERANDO ESCÁNERES EN C++

flex ofrece dos maneras distintas de generar analizadores para usar con C++. La primera manera es simplemente compilar un analizador generado por flex usando un compilador de C++ en lugar de un compilador de C. No debería encontrarse ante ningún error de compilación (por favor informe de cualquier error que encuentre a la dirección de correo electrónico dada en la sección Autores más abajo). Puede entonces usar código C++ en sus acciones de las reglas en lugar de código C. Fíjese que la fuente de entrada por defecto para su analizador permanece como yyin, y la repetición por defecto se hace aún a yyout. Ambos permanecen como variables FILE * y no como flujos de C++.

También puede utilizar flex para generar un analizador como una clase de C++, utilizando la opción -+ (o, equivalentemente, %option c++), que se especifica automáticamente si el nombre del ejecutable de flex finaliza con un '+', tal como flex++. Cuando se usa esta opción, flex establece por defecto la generación del analizador al fichero lex.yy.cc en vez de lex.yy.c. El analizador generado incluye el fichero de cabecera FlexLexer.h, que define el interfaz con las dos clases de C++.

La primera clase, FlexLexer, ofrece una clase base abstracta definiendo la interfaz a la clase del analizador general. Este provee las siguientes funciones miembro:

const char* YYText()

retorna el texto del token reconocido más recientemente, el equivalente a yytext.

int YYLeng()

retorna la longitud del token reconocido más recientemente, el equivalente a yyleng.

int lineno() const

retorna el número de línea de entrada actual (ver %option yylineno), o 1 si no se usó %option yylineno.

void set_debug(int flag)

activa la bandera de depuración para el analizador, equivalente a la asignación de yy_flex_debug (ver la sección Opciones más arriba). Fíjese que debe construir el analizador utilizando %option debug para incluir información de depuración en este.

int debug() const

retorna el estado actual de la bandera de depuración.

También se proveen funciones miembro equivalentes a yy_switch_to_buffer(), yy_create_buffer() (aunque el primer argumento es un puntero a objeto istream* y no un FILE*), yy_flush_buffer(), yy_delete_buffer(), y yyrestart() (de nuevo, el primer argumento es un puntero a objeto istream*).

La segunda clase definida en FlexLexer.h es yyFlexLexer, que se deriva de FlexLexer. Esta define las siguientes funciones miembro adicionales:

yyFlexLexer(istream* arg_yyin = 0, ostream* arg_yyout = 0)

construye un objeto yyFlexLexer usando los flujos dados para la entrada y salida. Si no se especifica, los flujos se establecen por defecto a cin y cout, respectivamente.

virtual int yylex()

hace el mismo papel que yylex() en los analizadores de flex ordinarios: analiza el flujo de entrada, consumiendo tokens, hasta que la acción de una regla retorne un valor. Si usted deriva una subclase S a partir de yyFlexLexer y quiere acceder a las funciones y variables miembro de S dentro de yylex(), entonces necesita utilizar %option yyclass="S" para informar a flex que estará utilizando esa subclase en lugar de yyFlexLexer. En este caso, en vez de generar yyFlexLexer::yylex(), flex genera S::yylex() (y también genera un substituto yyFlexLexer::yylex() que llama a yyFlexLexer::LexerError() si se invoca).

virtual void switch_streams(istream* new_in = 0, ostream* new_out = 0) reasigna yyin a new_in (si no es nulo) e yyout a new_out (idem), borrando el buffer de entrada anterior si se reasigna yyin.

int yylex(istream* new_in, ostream* new_out = 0)

primero conmuta el flujo de entrada via switch_streams(new_in, new_out) y entonces retorna el valor de yylex().

Además, yyFlexLexer define las siguientes funciones virtuales protegidas que puede redefinir en clases derivadas para adaptar el analizador:

virtual int LexerInput(char* buf, int max_size)

lee hasta max_size caracteres en buf y devuelve el número de caracteres leídos. Para indicar el fin-de-la-entrada, devuelve 0 caracteres. Fíjese que los analizadores "interactivos" (ver las banderas -B y -I) definen la macro YY_INTERACTIVE. Si usted redefine LexerInput() y

necesita tomar acciones distintas dependiendo de si el analizador está analizando una fuente de entrada interactivo o no, puede comprobar la presencia de este nombre mediante #ifdef.

virtual void LexerOutput(const char* buf, int size)

escribe a la salida size caracteres desde el buffer buf, que, mientras termine en NUL, puede contener también NUL's "internos" si las reglas del analizador pueden emparejar texto con NUL's dentro de este.

virtual void LexerError(const char* msg)

informa con un mensaje de error fatal. La versión por defecto de esta función escribe el mensaje al flujo cerr y finaliza.

Fíjese que un objeto yyFlexLexer contiene su estado de análisis completo. Así puede utilizar tales objetos para crear analizadores reentrantes. Puede hacer varias instancias de la misma clase yyFlexLexer, y puede combinar varias clases de analizadores en C++ conjuntamente en el mismo programa usando la opción -P comentada anteriormente.

Finalmente, note que la característica %array no está disponible en clases de analizadores en C++; debe utilizar %pointer (por defecto).

Aquí hay un ejemplo de un analizador en C++ simple:

// Un ejemplo del uso de la clase analizador en C++ de flex.

```
%{
int mylineno = 0;
%}

string \"[^\\n\"]+\\"
```

ws [\\t]+

```
alpha [A-Za-z]
dig [0-9]
name ({alpha}|{dig}|\\$)({alpha}|{dig}|[_\\.\\-\\/]*)*
num1 [-]?(dig)+\\.?([eE][-]?(dig)+)?
num2 [-]?(dig)*\\.{dig}+([eE][-]?(dig)+)?
number {num1}|{num2}
```

```
%%
```

```
{ws} /* evita los espacios en blanco y tabuladores */
```

```
\"/*"
{
int c;
```

```
while((c = yyinput()) != 0)
```

```

{
if(c == '\n')
    ++mylineno;

else if(c == '*')
{
    if((c = yyinput()) == '/')
        break;
    else
        unput(c);
}
}

{number} cout << "número " << YYText() << '\n';

\n      mylineno++;

{name}  cout << "nombre " << YYText() << '\n';

{string} cout << "cadena " << YYText() << '\n';

% %

```

```

int main( int /* argc */, char/**/ /* argv */ )
{
    FlexLexer* lexer = new yyFlexLexer;
    while(lexer->yylex() != 0)
        ;
    return 0;
}

```

Si desea crear varias (diferentes) clases analizadoras, use la bandera **-P** (o la opción **prefix=**) para renombrar cada **yyFlexLexer** a algún otro **xxFlexLexer**. Entonces puede incluir **<FlexLexer.h>** en los otros ficheros fuente una vez por clase analizadora, primero renombrando **yyFlexLexer** como se presenta a continuación:

```

#undef yyFlexLexer
#define yyFlexLexer xxFlexLexer
#include <FlexLexer.h>

#undef yyFlexLexer
#define yyFlexLexer zzFlexLexer
#include <FlexLexer.h>

```

si, por ejemplo, usted utilizó **%option prefix="xx"** para uno de sus analizadores y **%option prefix="zz"** para el otro.

IMPORTANTE: la forma actual de la clase analizadora es experimental y podría cambiar considerablemente entre versiones principales.

INCOMPATIBILIDADES CON LEX Y POSIX

flex es una reescritura de la herramienta lex del Unix de AT&T (aunque las dos implementaciones no comparten ningún código), con algunas extensiones e incompatibilidades, de las que ambas conciernen a aquellos que desean escribir analizadores aceptables por cualquier implementación. Flex sigue completamente la especificación POSIX de lex, excepto que cuando se utiliza %pointer (por defecto), una llamada a unput() destruye el contenido de yytext, que va en contra de la especificación POSIX.

En esta sección comentaremos todas las áreas conocidas de incompatibilidades entre flex, lex de AT&T, y la especificación POSIX.

La opción -l de flex activa la máxima compatibilidad con la implementación original de lex de AT&T, con el coste de una mayor pérdida de rendimiento en el analizador generado.

Indicamos más abajo qué incompatibilidades pueden superarse usando la opción -l.

flex es totalmente compatible con lex con las siguientes excepciones:

- La variable interna del analizador de lex sin documentar yylineno no se ofrece a menos que se use -l o %option yylineno.

yylineno debería gestionarse por buffer, en lugar de por analizador (simple variable global).

yylineno no es parte de la especificación POSIX.

- La rutina input() no es redefinible, aunque podría invocarse para leer los caracteres que siguen a continuación de lo que haya sido reconocido por una regla. Si input() se encuentra con un fin-de-fichero se realiza el procesamiento de yywrap() normal. input() retorna un fin-de-fichero ``real'' como EOF.

La entrada en su lugar se controla definiendo la macro YY_INPUT.

La restricción de flex de que input() no puede redefinirse va de acuerdo a la especificación POSIX, que simplemente no especifica ninguna manera de controlar la entrada del analizador que no sea haciendo una asignación inicial a yyin.

- La rutina unput() no es redefinible. Esta restricción va de acuerdo a POSIX.
- Los analizadores de flex no son tan reentrantes como los analizadores de lex. En particular, si tiene un analizador interactivo y un gestor de interrupción con long-jumps fuera del analizador, y el analizador a continuación se invoca de nuevo, podría obtener el siguiente mensaje:

fatal flex scanner internal error--end of buffer missed

Para volver al analizador, primero utilice

yyrestart(yyin);

Vea que esta llamada eliminará cualquier entrada en el buffer; normalmente esto no es un problema con un analizador interactivo.

Dese cuenta también de que las clases analizadoras en C++ son reentrantes, así que si usar C++ es una opción para usted, debería utilizarla. Vea "Generando Escáneres en C++" más arriba para los detalles.

- output() no se provee. La salida desde la macro ECHO se hace al puntero de fichero yyout (por defecto a stdout).

output() no es parte de la especificación POSIX.

- lex no acepta condiciones de arranque exclusivas (%x), aunque están en la especificación POSIX.

- Cuando se expanden las definiciones, flex las encierra entre paréntesis. Con lex, lo siguiente:

```
NOMBRE [A-Z][A-Z0-9]*  
% %  
foo{NOMBRE}? printf( "Lo encontró\n" );  
% %
```

no reconocerá la cadena "foo" porque cuando la macro se expanda la regla es equivalente a "foo[A-Z][A-Z0-9]*?" y la precedencia es tal que el '?' se asocia con "[A-Z0-9]*". Con flex, la regla se expandirá a

"foo([A-Z][A-Z0-9]*)?" y así la cadena "foo" se reconocerá.

Fíjese que si la definición comienza con ^ o finaliza con \$ entonces no se expande con paréntesis, para permitir que estos operadores aparezcan en las definiciones sin perder su significado especial. Pero los operadores <s>, /, y <<EOF>> no pueden utilizarse en una definición de flex.

El uso de -l produce en el comportamiento de lex el no poner paréntesis alrededor de la definición.

La especificación de POSIX dice que la definición debe ser encerrada entre paréntesis.

- Algunas implementaciones de lex permiten que la acción de una regla comience en una línea separada, si el patrón de la regla tiene espacios en blanco al final:

```
% %  
foobar<espacio aquí>  
{ foobar_action(); }
```

flex no dispone de esta propiedad.

- La opción %r de lex (generar un analizador Ratfor) no se ofrece. No es parte de la especificación de POSIX.

- Después de una llamada a unput(), el contenido de yytext está indefinido hasta que se reconozca el próximo token, a menos que el analizador se haya construido usando %array. Este no es el caso de lex o la especificación de POSIX. La opción -l elimina esta incompatibilidad.

- La precedencia del operador {} (rango numérico) es diferente. lex interpreta "abc{1,3}" como "empareja uno, dos, o tres apariciones de 'abc'", mientras que flex lo interpreta como "empareja 'ab' seguida de una, dos o tres apariciones de 'c'". Lo último va de acuerdo con la especificación de POSIX.

- La precedencia del operador ^ es diferente. lex interpreta "^foobar" como "empareja bien 'foo' al principio de una línea, o 'bar' en cualquier lugar", mientras que flex lo interpreta como "empareja 'foo' o 'bar' si vienen al principio de una línea". Lo último va de acuerdo con la especificación de POSIX.

- Las declaraciones especiales del tamaño de las tablas tal como %a que reconoce lex no se requieren en los analizadores de flex; flex los ignora.

- El identificador FLEX_SCANNER se #define de manera que los analizadores podrían escribirse para ser procesados con flex o con lex. Los analizadores también incluyen YY_FLEX_MAJOR_VERSION y YY_FLEX_MINOR_VERSION indicando qué versión de flex

generó el analizador (por ejemplo, para la versión 2.5, estas definiciones serán 2 y 5 respectivamente).

Las siguientes propiedades de flex no se incluyen en lex o la especificación POSIX:

analizadores en C++
%option
ámbitos de condiciones de arranque
pilas de condiciones de arranque
analizadores interactivos/no-interactivos
yy_scan_string() y sus amigas
yyterminate()
yy_set_interactive()
yy_set_bol()
YY_AT_BOL()
<<EOF>>
<*>
YY_DECL
YY_START
YY_USER_ACTION
YY_USER_INIT
directivas #line
%{}'s alrededor de acciones
varias acciones en una línea

más casi todas las banderas de flex. La última propiedad en la lista se refiere al hecho de que con flex puede poner varias acciones en la misma línea, separadas con punto y coma, mientras que con lex, lo siguiente

foo handle_foo(); ++num_foos_seen;

se trunca (sorprendentemente) a

foo handle_foo();

flex no trunca la acción. Las acciones que no se encierran en llaves simplemente se terminan al final de la línea.

DIAGNÓSTICOS

aviso, la regla no se puede aplicar indica que la regla dada no puede emparejarse porque sigue a otras reglas que siempre emparejarán el mismo texto que el de esta. Por ejemplo, en el siguiente ejemplo "foo" no puede emparejarse porque viene después de una regla "atrápalo-todo" para identificadores:

```
[a-z]+ obtuvo_identificador();
foo    obtuvo_foo();
```

El uso de REJECT en un analizador suprime este aviso.

aviso, se ha especificado la opción -s pero se puede aplicar la regla por defecto significa que es posible (tal vez únicamente en una condición de arranque en particular) que la regla por defecto (emparejar cualquier carácter simple) sea la única que emparejará una entrada particular. Ya que se indicó -s, presumiblemente esto no es lo que se pretendía.

definición no definida {reject_used_but_not_detected} o definición no definida {ymore_used_but_not_detected} - Estos errores pueden suceder en tiempo de compilación. Indican que el analizador usa REJECT o ymore() pero que flex falló en darse cuenta del hecho, queriendo decir que flex analizó las dos primeras secciones buscando apariciones de estas acciones y falló en encontrar alguna, pero que de algún modo se le han colado (por medio de un archivo #include, por ejemplo). Use %option reject o %option ymore para indicar a flex que realmente usa esta funcionalidad.

flex scanner jammed - un analizador compilado con -s ha encontrado una cadena de entrada que no fue reconocida por ninguna de sus reglas. Este error puede suceder también debido a problemas internos.

token too large, exceeds YYLMAX - su analizador usa %array y una de sus reglas reconoció una cadena más grande que la constante YYLMAX (8K bytes por defecto). Usted puede incrementar el valor haciendo un #define YYLMAX en la sección de definiciones de su entrada de flex.

el analizador requiere la opción -8 para poder usar el carácter 'x' - La especificación de su analizador incluye el reconocimiento del carácter de 8-bits 'x' y no ha especificado la bandera -8, y su analizador por defecto está a 7-bits porque ha usado las opciones -Cf o -CF de compresión de tablas. Vea el comentario de la bandera -7 para los detalles.

flex scanner push-back overflow - usted utilizó unput() para devolver tanto texto que el buffer del analizador no pudo mantener el texto devuelto y el token actual en yytext. Idealmente el analizador debería ajustar dinámicamente el buffer en este caso, pero actualmente no lo hace.

input buffer overflow, can't enlarge buffer because scanner uses REJECT - el analizador estaba intentando reconocer un token extremadamente largo y necesitó expandir el buffer de entrada. Esto no funciona con analizadores que usan REJECT.

fatal flex scanner internal error--end of buffer missed - Esto puede suceder en un analizador que se reintroduce después de que un long-jump haya saltado fuera (o sobre) el registro de activación del analizador. Antes de reintroducir el analizador, use:

yyrestart(yyin);

o, como se comentó más arriba, cambie y use el analizador como clase de C++.

too many start conditions in <> construct! - ha listado más condiciones de arranque en una construcción <> que las que existen (así que tuvo que haber listado al menos una de ellas dos veces).

FICHEROS

-lfl librería con la que los analizadores deben enlazarse.

lex.yy.c

analizador generado (llamado lexyy.c en algunos sistemas).

lex.yy.cc

clase generada en C++ con el analizador, cuando se utiliza -+.

<FlexLexer.h>

fichero de cabecera definiendo la clase base del analizador en C++, FlexLexer, y su clase derivada, yyFlexLexer.

flex.skl

esqueleto del analizador. Este fichero se utiliza únicamente cuando se construye flex, no cuando flex se ejecuta.

lex.backup

información de los retrocesos para la bandera -b (llamada lex.bck en algunos sistemas).

DEFICIENCIAS / ERRORES

Algunos patrones de contexto posterior no pueden reconocerse correctamente y generan mensajes de aviso ("contexto posterior peligroso"). Estos son patrones donde el final de la primera parte de la regla reconoce el comienzo de la segunda parte, tal como "zx*/xy*", donde el 'x*' reconoce la 'x' al comienzo del contexto posterior. (Fíjese que el borrador de POSIX establece que el texto reconocido por tales patrones no está definido.)

Para algunas reglas de contexto posterior, partes que son de hecho de longitud fija no se reconocen como tales, resultando en la pérdida de rendimiento mencionada anteriormente. En particular, las partes que usan 'l' o {n} (tales como "foo{3}") siempre se consideran de longitud variable.

La combinación de contexto posterior con la acción especial 'l' puede producir que el contexto posterior fijo se convierta en contexto posterior variable que es más caro. Por ejemplo, en lo que viene a continuación:

```
% %
abc  |
xyz/def
```

El uso de unput() invalida yytext e yyleng, a menos que se use la directiva %array o la opción -l.

La concordancia de patrones de NUL's es substancialmente más lento que el reconocimiento de otros caracteres.

El ajuste dinámico del buffer de entrada es lento, ya que conlleva el reanálisis de todo el texto reconocido hasta entonces por el (generalmente enorme) token actual.

Debido al uso simultáneo de buffers de entrada y lecturas por adelantado, no puede entremezclar llamadas a rutinas de <stdio.h>, tales como, por ejemplo, getchar(), con reglas de flex y esperar que funcione. Llame a input() en su lugar.

La totalidad de las entradas de la tabla listada por la bandera -v excluye el número de entradas en la tabla necesarias para determinar qué regla ha sido emparejada. El número de entradas es igual al número de estados del DFA si el analizador no usa REJECT, y algo mayor que el número de estados si se usa.

REJECT no puede usarse con las opciones **-f** o **-F**.

El algoritmo interno de flex necesita documentación.

VER TAMBIÉN

lex(1), yacc(1), sed(1), awk(1).

John Levine, Tony Mason, and Doug Brown, Lex & Yacc, O'Reilly and Associates. Esté seguro de obtener la 2^a edición.

M. E. Lesk and E. Schmidt, LEX - Lexical Analyzer Generator

Alfred Aho, Ravi Sethi and Jeffrey Ullman, Compilers: Principles, Techniques and Tools, Addison-Wesley (1986) (Edición en castellano: Compiladores: Principios, Técnicas y Herramientas, Addison-Wesley Iberoamericana, S.A. (1990)) Describe las técnicas de concordancia de patrones usadas por flex (autómata finito determinista).

AUTOR

Vern Paxson, con la ayuda de muchas ideas e inspiración de Van Jacobson. Versión original por Jef Poskanzer.

La representación de tablas rápidas es una implementación parcial de un diseño hecho por Van Jacobson. La implementación fue hecha por Kevin Gong y Vern Paxson.

Agradecimientos a los muchos flex beta-testers, feedbackers, y contribuidores, especialmente a Francois Pinard, Casey Leedom, Robert Abramovitz, Stan Adermann, Terry Allen, David Barker-Plummer, John Basrai, Neal Becker, Nelson H.F. Beebe, benson@odi.com, Karl Berry, Peter A. Bigot, Simon Blanchard, Keith Bostic, Frederic Brehm, Ian Brockbank, Kin Cho, Nick Christopher, Brian Clapper, J.T. Conklin, Jason Coughlin, Bill Cox, Nick Cropper, Dave Curtis, Scott David Daniels, Chris G. Demetriou, Theo Deraadt, Mike Donahue, Chuck Doucette, Tom Epperly, Leo Eskin, Chris Taylor, Chris Flatters, Jon Forrest, Jeffrey Friedl, Joe Gayda, Kaveh R. Ghazi, Wolfgang Glunz, Eric Goldman, Christopher M. Gould, Ulrich Grepel, Peer Griebel, Jan Hajic, Charles Hemphill, NORO Hideo, Jarkko Hietaniemi, Scott Hofmann, Jeff Honig, Dana Hudes, Eric Hughes, John Interrante, Ceriel Jacobs, Michal Jaegermann, Sakari Jalovaara, Jeffrey R. Jones, Henry Juengst, Klaus Kaempf, Jonathan I. Kamens, Terrence O Kane, Amir Katz, ken@ken.hilco.com, Kevin B. Kenny, Steve Kirsch, Winfried Koenig, Marq Kole, Ronald Lamprecht, Greg Lee, Rohan Lenard, Craig Leres, John Levine, Steve Liddle, David Loffredo, Mike Long, Mohamed el Lozy, Brian Madsen, Malte, Joe Marshall, Bengt Martensson, Chris Metcalf, Luke Mewburn, Jim Meyering, R. Alexander Milowski, Erik Naggum, G.T. Nicol, Landon Noll, James Nordby, Marc Nozell, Richard Ohnemus, Karsten Pahnke, Sven Panne, Roland Pesch, Walter Pelissero, Gaumond Pierre, Esmond Pitt, Jef Poskanzer, Joe Rahmeh, Jarmo Raiha, Frederic Raimbault, Pat Rankin, Rick Richardson, Kevin Rodgers, Kai Uwe Rommel, Jim Roskind, Alberto Santini, Andreas Scherer, Darrell Schiebel, Raf Schietekat, Doug Schmidt, Philippe Schnoebelen, Andreas Schwab, Larry Schwimmer, Alex Siegel, Eckehard Stolz, Jan-Erik Strvmquist, Mike Stump, Paul Stuart, Dave Tallman, Ian Lance Taylor, Chris Thewalt, Richard M. Timoney, Jodi Tsai, Paul Tuinenga, Gary Weik, Frank Whaley, Gerhard Wilhelms, Kent Williams, Ken Yap, Ron Zellar, Nathan Zelle, David Zuhn, y aquellos cuyos nombres han caído bajo mis escasas dotes de archivador de

correo pero cuyas contribuciones son apreciadas todas por igual.

Agradecimientos a Keith Bostic, Jon Forrest, Noah Friedman, John Gilmore, Craig Leres, John Levine, Bob Mulcahy, G.T. Nicol, Francois Pinard, Rich Salz, y a Richard Stallman por la ayuda con diversos quebraderos de cabeza con la distribución.

Agradecimientos a Esmond Pitt y Earle Horton por el soporte de caracteres de 8-bits; a Benson Margulies y a Fred Burke por el soporte de C++; a Kent Williams y a Tom Epperly por el soporte de la clase de C++; a Ove Ewerlid por el soporte de NUL's; y a Eric Hughes por el soporte de múltiples buffers.

Este trabajo fue hecho principalmente cuando yo estaba con el Grupo de Sistemas de Tiempo Real en el Lawrence Berkeley Laboratory en Berkeley, CA. Muchas gracias a todos allí por el apoyo que recibí.

Enviar comentarios a vern@ee.lbl.gov.

Sobre esta traducción enviar comentarios a Adrián Pérez Jorge (alu1415@csi.ull.es).

Versión 2.5

Abril 1995

FLEX(1)

Veamos un primer ejemplo:

Hacer un scanner que tome la entrada por teclado y reemplace: 'petrolio' por 'petróleo'.

Editamos un archivo denominado: **ejemplo-1.lex**

```
%{  
#include <stdio.h>  
%}  
%  
"petrolio"    printf("petróleo");  
.         printf("%s" , yytext );  
%  
int yywrap()  
{  
    return 1;  
}  
int main()  
{  
    yylex();  
    return 0; // Salida OK!  
}
```

Luego ejecutamos la orden:

flex -v ejemplo-1.lex

Que genera el archivo de Código fuente en Lenguaje C llamado: **lex.yy.c**

Y se compila con: **gcc -Wall lex.yy.c -o ejemplo-1.lex.out**

Que se ejecuta con: **./ejemplo-1.lex.out**

Vamos a analizar este archivo ejemplo-1.lex:

La EXPRESIÓN REGULAR es: “petrolio” y la ACCIÓN es: `printf("petróleo");`

El “.” significa “CUALQUIER OTRA EXPRESIÓN REGULAR”

yytext es una variable de flex, un puntero al string concreto, que siguió a la expresión regular.

Flex exige que haya definida una función denominada “yywrap”.

AUTÓMATAS FINITOS / EXPRESIONES REGULARES:

Usaremos esto para reconocer **PALABRAS CLAVE, SIGNOS, PUNTUACIONES**, etc.

Pero... hay un problema. Algunas construcciones sencillas dependen del contexto:

Ejemplo:

¿Hay diferencia? entre **gcc pepe.c** y **gcc pepe.c**
 // un blanco // 3 blancos

Depende... ¿en qué contexto se escribe esto?

gcc pepe.c y **gcc pepe.c** tienen el mismo efecto.

Pero

“gcc pepe.c” y “gcc pepe.c” no tienen el mismo efecto.

Así que un espacio en blanco depende de en contexto aparece.

Posibles soluciones:

a) Microscanners

Ejemplo:

¿Cómo hacemos para que un gran elefante pase desapercibido en la cancha de Rosario Central?

Pues, llenamos la cancha de elefantes.

Es decir, que esta técnica consiste en **tratar algunas expresiones con un scanner distinto**.

Ejemplo: Hacer un scanner que cuente espacios en blanco, considerando que N espacios en blanco sucesivos no entrecorbillados cuentan como uno solo, mientras que si están entre comillas cuentan como N espacios en blanco.

La especificación de este scanner sería la siguiente:

```
%{  
#include <stdio.h>  
static int cuantos = 0;  
%}  
%x COMILLA
```

% %

[]+ cuantos++;
"\""" BEGIN COMILLA;

```
<COMILLA>" "  cuantos++;  
<COMILLA>"\"""      BEGIN 0;  
"\\\"""      ;
```

% %

```
int yywrap(){ return 1; }
```

```
int main()
```

{

```
yylex();  
printf("cuantos = %d \n" , cuantos );  
return 0;
```

Esta especificación del scanner se guarda en un archivo llamado, por ejemplo, **cuenta-blancos.lex**
Entonces se lo procesa con: **flex -v cuenta-blancos.lex**
Con lo cuál se obtiene el archivo: **lex.yy.c**
Que se puede compilar con: **gcc -Wall lex.yy.c -o cuenta-blancos.lex.out**
Y se puede ejecutar con: **./cuenta-blancos.lex.out**

Este programa va a leer desde el teclado y va a contar los espacios sin entrecollar como un sólo espacio, y los espacios que estén entre comillas los cuenta a todos. Sólo va a detener su ejecución cuando reciba la señal de interrupción mediante teclado “Ctrl + D” y recién allí va a informar la cantidad de espacios contados bajo este criterio.

Vamos a ver ahora cómo implementar una calculadora para la línea de comandos. Para esto utilizaremos **Gramáticas Libres de Contexto**, es decir, vamos a usar a **flex junto con bison**. Editemos el archivo: calc.y que contiene la siguiente especificación del scanner junto con su gramática:

```
%{  
#include <stdio.h>  
#include <stdlib.h>  
#include <math.h>  
extern char * yytext;  
%}  
%token NRO SEN  
%union{    double d;    };  
%type<d> expr  
%left '+' '-'  
%left '*'  
%right UMINUS  
%%  
linea: expr ';'      { printf("%f\n", $1); YYACCEPT; }  
;  
expr: NRO          {$$=atof(yytext);};  
expr '+' expr      { $$ = $1 + $3; }  
expr '*' expr      { $$ = $1 * $3; }  
expr '-' expr      { $$ = $1 - $3; }  
!'-' expr %prec UMINUS { $$ = -$2; }  
!(' expr ')'       { $$ = $2; }  
!SEN '(' expr ')' { $$ = sin( $3 ); }  
;  
%%  
int yyerror( char * s )  
{    fprintf( stderr, "%s \n", s );  
    return 0;  
}  
int main()  
{  
    while( yyparse() )  
    ;  
    return 0;  
}
```

Este archivo **calc.y** se procesa con **bison** mediante la orden: **bison -d calc.y**

Esto genera dos archivos: **calc.tab.h** y **calc.tab.c**

Pero para que esto funcione como una calculadora, necesitaremos **asociar a la SINTAXIS una SEMÁNTICA**. Para describir esta **SEMÁNTICA** debemos asociar **ATRIBUTOS** a los **TERMINALES** y **NO-TERMINALES**. Declaramos estos atributos así:

```
%token NRO SEN
%union{    double d;    };
%type<d> expr
%left '+' '-'
%left '*'
%right UMINUS
```

Para acceder a estos atributos usamos los indicadores posicionales denotados con \$1, \$2, \$3 , etc según estén en la 1º, 2º, o 3º posición, en el lado derecho de una expresión y \$\$ en el lado izquierdo de la misma.

Ahora editemos el archivo que va a contener la especificación del scanner: **calc.lex**:

```
%{
#include "calc.tab.h"
%}
```

```
%%
```

```
[\\t\\n]+ ;
```

```
"("      return '(';
")"      return ')';
"*"      return '*';
"+"      return '+';
";"      return ';';
"sin"    return SEN;
[0-9]+   return NRO;
[0-9]+."[0-9]+ return NRO;
.        abort();
```

```
%%
```

```
int yywrap()
{
    return 1;
}
```

Este archivo se procesa con **flex** mediante la orden: **flex -v calc.lex**

Generando el archivo: **lexx.yy.c**

El último paso que nos falta para generar el archivo ejecutable de la calculadora es el siguiente:

```
gcc -Wall calc.tab.c lex.yy.c -lm -o calculadora.out
```

generando el archivo ejecutable: **calculadora.out**

// Clases en Lenguaje c++ para el interprete de comandos:

// Definiciones de clases van en el archivo "clases.hpp"

```
#include <iostream.h>
enum modo { __Fork, __Exec }

// clase abstracta
class Accion {
public:
    virtual ~Accion(){}  

    virtual int exec( enum modo ) = 0;  

};
```

// clases concretas

```
class Cd : public Accion { // "Cd" extiende a "Accion"
    char * path;
public:
    Cd( char * path ) : path( path ) {} // constructor
    int exec( enum modo ) { chdir( path ); }
    ~Cd() { delete path; } // destructor
};
```

```
class Pwd : public Accion { // "Pwd" extiende a "Accion"
public:
    int exec( enum modo ) { cout << get_current_dir_name << '\n' ; return 0; }
};
```

```
class Comando : public Accion { // "Comando" extiende a "Accion"
    char *prog , **args;
    static char * path( char * );
public:
    Comando( char * p , char ** a ) : prog( path( p ) ), args(a) {} // Constructor
    ~Comando() { delete prog ; delete args; } // Destructor
// Ejercicio: acá hay un problema encontrarlo y subsanarlo ( "delete args" )
    int exec( enum modo);
};
```

// En el archivo “clases.cc”

```
char * Comando :: path( char * p)
{
    char * tmp = new char[ 1024 ];
    if( strchr( p , '/' ) != 0 ) {
        strcpy( tmp , p );
        return tmp;
    }
    char env[ 1024 ];
    strcpy( env , getenv("PATH") );
    char * pc = strtok( env , ":" );
    do{
        strcat( streat( strcpy( tmp , pc ) , "/" ) , p );
        if( acces( tmp, X_OK ) == 0 )
            return tmp;
    } while ( ( pc = strtok( NULL , ":" ) ) != 0 );
    return p;
}
```

// hay que agregar a los prototipos

```
//         #include <stdlib.h>
//         #include <unistd.h>
```

// ver: man fun.c

// ver: man string.h

// continúa código para el archivo “clases.cc”

int Comando :: exec(enum modo m)

```
{
    if( m == __Fork && fork() != 0 ) {
        int status;
        wait( &status );
        return status;
    }
    glob_k pg;
    int r;
    if( ( r = glob( arg[0] , GLOB_TILDE | GLOB_NOCHECK , NULL, &pg ) ) )
        glob_error( r );
    for( int i=1 ; arg[ i ] , i++ )
        if( ( r = glob( args[i], GLOB_TILDE | GLOB_NOCHECK | GLOB_APPEND ,
                        NULL, &pg ) ) )
            glob.error( r );
    execv( prog , pg.gl_pathv );
    exit( -1 );
    return 0;
}
```

```
// Editar y poner en “clases.hpp” todas las definiciones de las clases
// y el archivo clases.cc como se indica en comentario
// con lo que se tiene corregir detalles
// ver glob.error( int e )
// ver: man glob
// agregar un main
```

```
// main de ejemplo para el archivo “clases.cc”
int main()
{
    char ** args = { “ls” , “*.c*” , “a*” , NULL }
    Comando c( “ls” , args );
    c.exec( __Exec );
    return 0;
}
```