



Laboratorio Docente de Computación

## Guia Básica del Lenguaje C ANSI

<ldc@ldc.usb.ve>

Enero 2007

# Índice

<b>1. Antecedentes</b>	<b>3</b>
1.1. Descarga e instalación . . . . .	5
1.1.1. Windows . . . . .	5
1.1.2. Linux . . . . .	5
1.2. Editores . . . . .	6
1.2.1. Windows . . . . .	6
1.2.2. Linux . . . . .	6
<b>2. Introducción al lenguaje C</b>	<b>7</b>
<b>3. Tipos, operadores y expresiones</b>	<b>9</b>
3.1. Tipos y tamaños de datos . . . . .	9
3.2. Conversión de tipos . . . . .	10
3.3. Expresiones y operadores . . . . .	12
3.4. Control de flujo . . . . .	14
3.4.1. Sentencia if . . . . .	14
3.4.2. Setencia switch . . . . .	14
3.4.3. Setencia while . . . . .	15
3.4.4. Setencia for . . . . .	15
3.4.5. Break y Continue . . . . .	16
3.4.6. Goto y etiquetas . . . . .	17
<b>4. Estructuras de Datos</b>	<b>18</b>
4.1. Ejemplo Básico . . . . .	18
4.2. Estructuras y Funciones . . . . .	19
4.3. Arreglos de Estructuras . . . . .	21
4.4. Apuntadores a Estructuras . . . . .	24
4.5. Typedef . . . . .	26
<b>5. Entrada/Salida</b>	<b>27</b>
5.1. Manejo de Salida y Entrada de Texto - Printf y Scanf . . . . .	28
5.2. Manejo de Archivos . . . . .	30
5.3. Funciones Misceláneas . . . . .	34
<b>6. Apuntadores</b>	<b>37</b>
6.1. ¿Qué es un apuntador? . . . . .	37
6.2. Apuntadores y estructuras de datos . . . . .	40
6.2.1. Arreglos . . . . .	40
6.2.2. Lista enlazada . . . . .	44

6.3. Apuntadores y strings . . . . .	46
<b>7. Funciones</b>	<b>48</b>
<b>8. Estructura de un programa</b>	<b>53</b>
8.1. Variables externas . . . . .	53
8.2. Archivos de Encabezado . . . . .	58
8.3. Estructura de Bloques . . . . .	59
8.4. Inicialización . . . . .	60
8.5. Recursión . . . . .	60
8.6. El preprocesador de C . . . . .	62
8.6.1. Inclusión de Archivos . . . . .	62
8.6.2. Substituciones Macro . . . . .	62
8.6.3. Inclusión Condicional . . . . .	63
<b>9. Herramientas</b>	<b>65</b>
9.1. Makefile . . . . .	65
9.2. <i>Debugging</i> con gdb . . . . .	68
9.3. Core dumps . . . . .	69
<b>10. Ejercicios</b>	<b>70</b>

# 1. Introducción

C es un lenguaje de programación creado en 1969 por Ken Thompson y Dennis M. Ritchie en los Laboratorios Bell como evolución del lenguaje B, a su vez basado en BCPL. Al igual que B, es un lenguaje orientado a la implementación de Sistemas Operativos, concretamente Unix. C es apreciado por la eficiencia del código que produce y es el lenguaje de programación más popular para crear software de sistemas, aunque también se utiliza para crear aplicaciones. El desarrollo inicial de C se llevó a cabo en los Laboratorios Bell de AT&T entre 1969 y 1973; según Ritchie, el período más creativo tuvo lugar entre 1969 y 1973 y en 1972.

La principal estandarización del lenguaje C es la conocida como “ANSI C”, con el estándar X3.159-1989. Posteriormente, en 1990, éste fue ratificado como estándar ISO (ISO/IEC 9899:1990). La adopción de este estándar es muy amplia, por lo que los programas que lo siguen corren sobre una gran variedad de plataformas. En la práctica, los programadores suelen usar elementos no portables dependientes del compilador o del sistema operativo.

En 1973, el lenguaje C se había vuelto tan poderoso que la mayor parte del núcleo del sistema operativo Unix, originalmente escrito en el lenguaje ensamblador PDP-11/20, fue reescrita en C. Éste fue uno de los primeros núcleos de sistema operativo implementados en un lenguaje distinto al ensamblador; algunos casos anteriores son el sistema Multics, escrito en PL/I, y Master Control Program para el B5000 de Burroughs, escrito en Algol en 1961.

En 1978, Ritchie y Brian Kernighan publicaron la primera edición de “El Lenguaje de programación C”. Éste libro fue durante años la especificación informal del lenguaje. El lenguaje descrito en este libro recibe habitualmente el nombre de “el C de Kernighan y Ritchie” o simplemente “K&R C”. La segunda edición del libro cubre el estándar ANSI C.

El C de Kernighan y Ritchie es el subconjunto más básico del lenguaje que un compilador debe de soportar. Durante muchos años, incluso tras la introducción del ANSI C, fue considerado “el mínimo común denominador” en el que los programadores debían programar cuando deseaban que sus programas fueran portables, pues no todos los compiladores soportaban completamente ANSI, y el código razonablemente bien escrito en K&R C es también código ANSI C válido.

A finales de la década de 1970, C empezó a sustituir a BASIC como el lenguaje de programación de microcomputadores predominante. Durante la década de 1980 se empezaron a usar los IBM PC, lo que incrementó su popularidad significativamente. Al mismo tiempo, Bjarne Stroustrup empezó a trabajar con algunos compañeros de Bell Labs para añadir funcionalidades

de programación orientada a objetos a C. El lenguaje que crearon, llamado C++, es hoy en día el lenguaje de programación de aplicaciones más común en el sistema operativo Microsoft Windows; C sigue siendo más popular en el entorno Unix.

En 1983, el Instituto Nacional Estadounidense de Estándares (ANSI, por sus siglas en inglés) organizó un comité, X3j11, para establecer una especificación estándar de C. Tras un proceso de trabajo largo y arduo, se completó el estándar en 1989 y se ratificó como el "Lenguaje de Programación C" ANSI X3. 159-1989. Esta versión del lenguaje se conoce a menudo como ANSI C, o a veces como C89 (para distinguirla de C99).

En 1990, el estándar ANSI (con algunas modificaciones menores) fue adoptado por la Organización Internacional para la Estandarización (ISO) en el estándar ISO/IEC 9899:1990. Esta versión se conoce a veces como C90. No obstante, "C89" y "C90" se refieren en esencia el mismo lenguaje.

Uno de los objetivos del proceso de estandarización del ANSI C fue producir una extensión al C de Kernighan y Ritchie, incorporando muchas funcionalidades no oficiales y también nuevas.

ANSI C está soportado hoy en día por casi la totalidad de los compiladores de C. La mayoría del código C que se escribe actualmente está basado en ANSI C. Cualquier programa escrito sólo en C estándar sin código que dependa de un hardware determinado funciona correctamente en cualquier plataforma que disponga de una implementación de C compatible. Sin embargo, muchos programas han sido escritos de forma que sólo pueden compilarse en una cierta plataforma, o con un compilador concreto, debido a la utilización de librerías no estándar, como interfaces gráficas de usuario. Además, algunos compiladores no cumplen, en el modo por defecto, las especificaciones del estándar ANSI C o su sucesor. Por último, ocurre frecuentemente que el código está escrito con dependencia de un tamaño determinado de ciertos tipos de datos, o de un determinado orden de los bits en la memoria principal de la plataforma.

## 2. Tipos, operadores y expresiones

De manera resumida, podemos decir que las variables y las constantes son los contenedores de datos básicos que se manipulan en un programa. La sección de declaración de un programa define las variables que se van a utilizar, su tipo y, opcionalmente, sus valores iniciales. El tipo de un objeto determina el conjunto de valores que puede almacenar y las operaciones que se pueden realizar sobre él. También, Los operadores especifican lo que se hará con las variables y las expresiones combinan variables y constantes para producir nuevos valores.

### 2.1. Tipos y tamaños de datos

A continuación tenemos una tabla con los tipos básicos que podemos encontrar en el lenguaje C:

char	Caracter
short	Entero corto
int	Entero
long	Entero largo
unsigned	Entero sin signo
float	Número con coma flotante
double	Número con coma flotante de doble precisión

El tamaño en bits de cada tipo depende de la arquitectura y el tipo de software que corre sobre la máquina en que ejecutamos un programa. Una forma fácil de conocer el tamaño de cada tipo en una máquina dada es compilar y ejecutar sobre ella un programa como el siguiente. Éste hace uso del operador **sizeof**, que nos da el número de bytes que ocupa un tipo dado.

```
#include <stdio.h>

int main() {
    printf("Tamano de char: %d\n", sizeof(char));
    printf("Tamano de short: %d\n", sizeof(short));
    printf("Tamano de int: %d\n", sizeof(int));
    printf("Tamano de long: %d\n", sizeof(long));
    printf("Tamano de float: %d\n", sizeof(float));
    printf("Tamano de double: %d\n", sizeof(double));
}
```

Código 1: Tamaños de tipos distintos en C

Podemos especificar *modificadores* para los tipos al momento de declarar una variable. Éstos pueden alterar el tamaño del tipo o su comportamiento. Por ejemplo, la palabra **unsigned** en realidad es un modificador aplicable a cualquier tipo entero, aunque por sí sola se refiere al tipo **int**. Su efecto es quitarle el signo al número, lo que nos da un mayor rango de números positivos representables.

Hay ciertas relaciones que podemos asumir entre los tipos de C, independientemente de la plataforma sobre la que estamos trabajando. El tipo **long** tiene al menos 32 bits y **short** nunca es mayor que **int**, el cual a su vez nunca es mayor que **long**. Los calificadores **signed** y **unsigned** pueden aplicarse a un **char** y a cualquier entero. Los números **unsigned** son siempre no negativos y obedecen las leyes de la aritmética módulo  $2n$ , donde  $n$  es el número de bits en el tipo.

Las variables son definidas utilizando un identificador de tipo seguido del nombre de la variable. Veamos el siguiente programa:

```
#include <stdio.h>

main() {
    float cels , farh;
    farh = 35.0;
    cels = 5.0 * ( farh - 32.0 ) / 9.0;
    printf("-> %f F son %f C\n" , farh , cels );
}
```

Código 2: Conversión de temperatura de grados Fahrenheit a Celsius

Aquí podemos ver que se definen dos variables float, se asigna un valor a la primera y se calcula la segunda mediante una expresión aritmética. En C, las asignaciones también son expresiones y se pueden utilizar como parte de otra expresión; sin embargo, prácticas de este tipo no son muy recomendables ya que reducen la legibilidad del programa. En la instrucción printf, el *especificador* %f indica que se quiere imprimir un número con coma flotante.

Además de todo esto hay un tipo muy importante llamado void, que representa algo “vacío” y tiene uso al escribir funciones (cuando éstas no devuelven un valor) y al utilizar apuntadores (cuando se quiere tener un apuntador a una variable de tipo desconocido). Ambos usos serán descritos en las secciones de funciones y apuntadores respectivamente.

## 2.2. Conversión de tipos

Cuando escribimos una expresión aritmética en la cual hay variables o valores de distintos tipos, el compilador realiza determinadas conversiones antes de evaluarla. Estas conversiones pueden aumentar o disminuir la precisión del tipo al que se convierten los elementos de la expresión. Un ejemplo claro es la comparación de una variable de tipo **int** con una variable de tipo **double**. En este caso, la de tipo **int** es convertida a **double** para poder realizar la comparación. Estas conversiones son comúnmente conocidas con el nombre de *cast*.

Regularmente, se pueden mezclar distintos tipos de valores en expresiones aritméticas. Por ejemplo, el tipo `char` puede ser tratado como entero. Sin embargo, cuando se quieren realizar operaciones con tipos de distintos tamaños, el resultado será del tamaño del más grande. Las operaciones entre un número con punto flotante de precisión simple y uno de precisión doble, el resultado tiene tipo **double**. Usualmente, no hay problemas en asignar valores entre variables de diferentes tipos, pero existen algunas excepciones:

- Cuando la variable es demasiado pequeña como para guardar el valor asignado, éste se corrompe.
- Cuando a una variable de tipo entero se le asigna un valor con decimales. En este caso, la máquina normalmente redondea y se pierde la precisión.

También podemos realizar conversiones explícitas entre tipos. Tomemos como ejemplo el siguiente trozo de código:

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

main()
{
    int i = 256;
    int root
    root = sqrt( (double) i );
    printf("Resultado : %d\n", root);
}
```

Código 3: Raíz cuadrada de un número entero

Este programa calcula la raíz cuadrada de un número entero, previa conversión a tipo **double**. La conversión es realizada colocando entre paréntesis

el nombre del tipo requerido justo antes del valor. En este caso, `(double)`, el resultado de `sqrt( (double) i)`; es también un doble, pero es convertido automáticamente a entero en la asignación a la variable `root`.

Otro ejemplo se presenta a continuación:

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

int main() {

    float a;
    int b;
    b = 10;
    a = 0.5;
    if ( a <=(float) b ) {
        b = (float) a;
    }
}
```

Código 4: Ejemplo de conversión

### 2.3. Expresiones y operadores

Los distintos operadores del lenguaje permiten formar expresiones tanto aritméticas como lógicas:

+, -	suma, resta
++, --	incremento, decremento
, /, %	multiplicación, división, módulo
«, »	rotación de bits a la derecha, izquierda
&	AND booleano
	OR booleano
~	complemento a 1
!	complemento a 2, NOT lógico
==, !=	igualdad, desigualdad
&&,	AND, OR lógico
<, <=	menor, menor o igual
>, >=	mayor, mayor o igual

Al usarlos debe tenerse en cuenta su precedencia y las reglas de asociatividad, que son las normales en la mayoría de lenguajes y en matemáticas. Se debe consultar el manual de referencia para obtener una explicación detallada. Además hay toda una serie de operadores que realizan una operación

y una asignación a la vez, como += y &=.

En la evaluación de expresiones lógicas, los compiladores normalmente utilizan técnicas de evaluación rápida. Para decidir si una expresión lógica es cierta o falsa muchas veces no es necesario evaluarla completamente. Por ejemplo, dada una expresión como <exp1> || <exp2>, el compilador evalúa primero <exp1> y, si es cierta, no evalúa <exp2>. Por ello se deben evitar construcciones en las que se modifiquen valores de datos en la propia expresión: su comportamiento puede depender de la implementación del compilador o de la optimización utilizada en una compilación o en otra. Estos son errores que se pueden cometer fácilmente en C ya que una asignación es también una expresión.

Debemos evitar instrucciones como

```
if (( x++ > 3 ) || ( x < y ))
```

y escribir en su lugar

```
x++; if (( x > 3 ) || ( x < y ))
```

Hay un tipo especial de expresión en C que se denomina expresión condicional y está representada por los operadores “? :”. Se utiliza así:

```
<e> ? <x> : <y>
```

Esta operación da como resultado <x> si <e> es cierto e <y> si no. A continuación, un ejemplo de su uso:

```
#include <math.h>

int main()
{
    int a = 1, b = 3;
    float x, y = 0.0;
    int PI = 3.14159;

    x = (a > b) ? a : b;
    (b*a > 0.0) ? (y=sin(PI / 8)) : (y=cos(PI / 4));
}
```

Código 5: Expresiones condicionales

## 2.4. Control de flujo

Para alterar el flujo de ejecución del programa, que por sí sólo consiste en la ejecución de todas las instrucciones paso a paso en el orden en que fueron dadas, usamos las llamadas “sentencias de control de flujo”. Ellas nos permiten tomar decisiones y realizar calculos mucho más complejos.

### 2.4.1. Sentencia if

La sentencia de control de decisión básica es `if (<e>) then <s> else <t>`. Evalúa una expresión booleana y, si se cumple, ejecuta la sentencia `s`. En caso contrario, ejecuta la sentencia `t`. La segunda parte de sentencia `–else <t>`–, es opcional.

```
void cero( double a ) {
    if ( a == 0.0 )
        printf("a es cero\n");
    else
        printf("a es diferente de cero\n");
}
```

Código 6: Control de flujo con **if**

Recordemos que C considera valores diferentes de cero como “verdaderos” y el cero como “falso”.

### 2.4.2. Sentencia switch

La sentencia `switch` es útil cuando se quiere comparar un valor dado contra un rango grande de valores. Se usa de la siguiente manera:

```
switch( valor ) {
    case valor1: <sentencias>
    case valor2: <sentencias>
    ...
    default: <sentencias>
}
```

Código 7: Control de flujo con **switch**

Cuando se encuentra una sentencia `case` que concuerda con el valor del `switch` se ejecutan las sentencias que le siguen y todas las demás a partir

de ahí, hasta conseguir la primera instrucción **break**. La entrada **default** es opcional; se activa cuando ningún otro caso de comparación resulta cierto.

```
ver_opcion( char c ) {  
  
    switch(c) {  
        case 'a':  
            printf("Op A\n");  
            break;  
        case 'b':  
            printf("Op B\n");  
            break;  
        case 'c':  
        case 'd':  
            printf("Op C o D\n");  
            break;  
        default:  
            printf("Op ?\n");  
    }  
}
```

Código 8: Ejemplo de **switch**

### 2.4.3. Sentencia **while**

Otra clase de sentencias de control de flujo nos permite realizar repeticiones de una serie de instrucciones. En C tenemos tres sentencias de repetición. Para empezar tenemos la sentencia **while** (<e>) <s> Las instrucciones <s> se ejecutan repetidamente mientras la evaluación de la expresión <e> sea verdadera.

```
long raiz( long valor ) {  
    long r = 1;  
    while ( r * r <= valor )  
        r++;  
    return r;  
}
```

Código 9: Control de flujo con **while**

Una variación ligera de la sentencia **while** es **do ... while**, que se escribe así: **do** <s> **while** { <e> }. Ésta ejecuta el conjunto de instrucciones dado al menos una vez. Es decir, sólo evalúa la expresión condicional en la segunda repetición.

#### 2.4.4. Sentencia for

Otra sentencia iterativa, que permite inicializar los controles del ciclo es la sentencia **for**:

```
for (<i>; <e>; <p>) {  
    <s>  
}
```

Al escribir esto:

1. Se ejecuta *i*
2. Mientras *<e>* es cierto, se ejecuta *<s>* y luego *<p>*.

Notemos que esto es equivalente a

```
<i>;  
while ( <e> ) {  
    <s>;  
    <p>;  
}
```

El ejemplo anterior se podría escribir como:

```
long raiz( long valor ) {  
    long r;  
    for(r = 1; r*r <= valor; r++);  
    return r;  
}
```

Código 10: Control de flujo con **for**

#### 2.4.5. Break y Continue

Las instrucciones **break** y **continue** son utilizadas dentro de sentencias de repetición para alterar el flujo de control dentro de las mismas. **break** provoca que se termine la ejecución de una iteración. **continue** provoca que se comience una nueva iteración, saltándose las instrucciones restantes de la repetición y evaluando la expresión de control. Veamos el siguiente ejemplo:

```

void final_countdown(void) {
    int count = 10;
    while( count > 1 ) {
        if( count == 4 )
            start_engines();

        if( status() == WARNING )
            break;

        printf("%d ", count );
    }

    if( count == 0 ) {
        launch();
        printf("Shuttle launched\n");
    }
    else
    {
        printf("WARNING condition received.\n");
        printf("Count held at T - %d\n", count );
    }
}

```

Código 11: Control de flujo con **break**

### 3. Estructuras de Datos

Una estructura es una colección de una o más variables, posiblemente de diferentes tipos, agrupadas bajo un nombre que permite hacer referencia a éstas. Las estructuras ayudan a organizar datos complicados, particularmente en programas largos, ya que permite reunir un grupo de variables relacionadas para que sean tratadas como una unidad en lugar de distantes entidades.

Un ejemplo tradicional es la nómina de un empleado, un empleado se puede describir con un nombre, dirección, número de seguro social, cédula de identidad, salario, etc. Incluso alguno de éstos podrían ser estructuras a su vez. Igualmente, se puede considerar un punto en el eje de coordenadas como una estructura, al igual que un rectángulo es un conjunto de puntos, y así sucesivamente.

#### 3.1. Ejemplo Básico

Continuando el ejemplo de los puntos en una recta, se puede definir un punto mediante la siguiente estructura:

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

La palabra clave *struct* introduce la declaración de una estructura, que es una lista de declaraciones entre llaves. Este tipo de declaración define un tipo. La llave que cierra la estructura puede estar seguida de una lista de variables, como cualquier otro tipo.

```
struct{...} x, y, z;
```

Esto es sintácticamente análogo a:

```
int x, y, z;
```

En el sentido que cada declaración establece a estas variables como del tipo especificado, `struct` e `int` y el espacio correspondiente es asignado para ellas.

Una declaración que no es seguida de una lista de variables no reserva espacio en memoria, ya solo describe una plantilla de una estructura: `struct x;`. Sin embargo, en el caso `struct point pt;` se define una variable `pt` que es una estructura de tipo *struct point*.

Una estructura puede ser inicializada especificando una lista de expresiones constantes para cada miembro, `struct maxpt = {320,200};`. Igualmente, una estructura puede inicializarse por asignación o por la llamada de una función que devuelve el tipo correcto.

Los miembros de una estructura se acceden por una construcción de la forma *nombre-estructura.nombre-miembro*. El punto "." conecta el nombre de la estructura con el nombre del miembro. Por ejemplo para imprimir las coordenadas se utiliza la instrucción `printf( '%d,%d', pt.x, pt.y);`.

Las estructuras pueden ser anidadas, por ejemplo:

```
struct rect {
    struct point pt1;
    struct point pt2;
};
```

Si se declara *screen* como un tipo *rect*, para referirse a la coordenada x del primer punto es *screen.pt1.x*.

## 3.2. Estructuras y Funciones

Se utiliza un ejemplo para ilustrar mejor al lector:

```
/* makepoint: construye un punto
de las componente x e y */
struct point makepoint(int x, int y)
{
    struct point temp;
    temp.x = x;
    temp.y = y;
    return temp;
};
```

Esta función puede ser usada para inicializar cualquier estructura dinámicamente, o para proveer argumentos para una función:

```
struct rect screen;
struct point middle;
struct point makepoint(int, int);
screen.pt1 = makepoint(0,0);
screen.pt2 = makepoint(XMAX, YMAX);
middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
                  (screen.pt1.y + screen.pt2.y)/2);
```

El próximo paso es hacer una función que aplique aritmética entre dos puntos:

```

/* addpoints: add two points */
struct addpoint(struct point p1, struct point p2)
{
p1.x += p2.x;
p1.y += p2.y;
return p1;
}

```

Aquí tanto los argumentos como el valor de retorno son estructuras. Se incrementan los componentes en *p1* en lugar de usar una variable temporal con el fin de enfatizar que las estructuras son pasadas como parámetros como cualquier otro valor.

Si una estructura grande se quiere pasar a una función, generalmente es más eficiente pasar un apuntador que copiar toda la estructura. Apuntadores a estructuras son como un apuntador a cualquier tipo de variables. La declaración se da a continuación:

```
struct point *pp;
```

Indica que *pp* es un apuntador a una estructura de tipo *struct point*. Si *pp* apunta a una estructura *point*, *\*pp* es la estructura, y *(\*pp).x* y *(\*pp).y* son los miembros. Para usar *pp*, se puede escribir:

```

struct point origin, *pp;
pp = &origin;
printf('origen es (%d,%d)\n', (*pp).x, (*pp).y);

```

Los paréntesis son necesarios en el acceso a los miembros ya que el operador "." tiene mayor precedencia que "\*\*".

También es importante conocer que si *p* es un apuntador a una estructura, entonces *p->miembro-estructura*, se refiere a un miembro en particular, así que es equivalente escribir:

```
printf('origen es (%d, %d)\n', pp->x, pp->y);
```

Entonces, las siguientes expresiones son equivalentes:

```
struct rect r, *rp = &r;
```

```

r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x

```

### 3.3. Arreglos de Estructuras

A continuación se presenta un programa que cuenta las ocurrencias de palabras clave del lenguaje C leídas desde la entrada estándar. Se requiere un arreglo de strings para guardar los nombres y un arreglo de enteros que llevan contadores de cada palabra clave conseguida. Una posibilidad es usar dos arreglos paralelos, *keyboard* y *keycount*:

```
char *keyword[NKEYS];
int keycount[NKEYS];
```

Sin embargo, existe una mejor forma de organizar:

```
char *word;
int cout;

struct key {
    char *word;
    int count;
} keytab[NKEYS];
```

O también:

```
struct key {
    char *word;
    int count;
};
struct key keytab[NKEYS];
```

Esto declara una estructura tipo *key*, define un arreglo *keytab* de estructuras de este tipo, y establece el espacio en memoria necesario. Dado que *keytab* contiene un conjunto de nombres, la inicialización es trivial:

```
struct key {
    char *word;
    int count;
} keytab[] = {
    "auto", 0,
    "break", 0,
    "case", 0,
    "char", 0,
    "const", 0,
    "continue", 0,
```

```

    "default", 0,
    /* ... */
    "unsigned", 0,
    "void", 0,
    "volatile", 0,
    "while", 0
};

```

Es equivalente encerrar entre llaves cada elemento del arreglo cuando son tipos básicos:

```

{ "auto", 0 },
{ "break", 0 },
{ "case", 0 },
...

```

El programa comienza con la definición de *keytab*. La rutina *main* lee de la entrada repetidamente cada palabra a través de la función *getword*. Cada palabra es chequeada en el arreglo con una función que realiza una búsqueda binaria, por esto las claves deben estar ordenadas de forma creciente.

```

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100

int getword(char *, int);
int binsearch(char *, struct key *, int);

/* count C keywords */
main()
{
    int n;
    char word[MAXWORD];
    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((n = binsearch(word, keytab, NKEYS)) >= 0)
                keytab[n].count++;
    for (n = 0; n < NKEYS; n++)
        if (keytab[n].count > 0)
            printf("%4d %s\n",
                keytab[n].count,

```

```

        keytab[n].word);
    return 0;
}

/* binsearch: find word in tab[0]...tab[n-1] */
int binsearch(char *word, struct key tab[], int n)
{
    int cond;
    int low, high, mid;
    low = 0;
    high = n - 1;
    while (low <= high)
    {
        mid = (low+high) / 2;
        if ((cond = strcmp(word, tab[mid].word)) < 0)
            high = mid - 1;
        else if (cond > 0)
            low = mid + 1;
        else
            return mid;
    }
    return -1;
}

/* getword: get next word or character from input */
int getword(char *word, int lim)
{
    int c, getch(void);
    void ungetch(int);
    char *w = word;
    while (isspace(c = getch()))
        ;
    if (c != EOF)
        *w++ = c;
    if (!isalpha(c))
    {
        *w = '\0';
        return c;
    }
    for ( ; --lim > 0; w++)
        if (!isalnum(*w = getch()))

```

```

    {
        ungetch(*w);
        break;
    }
    *w = '\0';
    return word[0];
}

```

Código 12: Keyword Count

El tamaño de este arreglo se puede obtener a través de la multiplicación de la cantidad de entradas por el tamaño de cada una. La cantidad de elementos es conocida por el programador en este caso, y el tamaño de cada elemento puede ser dado por la función `sizeof (type name)`. Esta función retorna el tamaño de un tipo en bytes.

La macro `#define NKEYS (sizeof keytab / sizeof(keytab[0]))` puede ser usada para darle un valor a `NKEYS`. Usualmente, la función `sizeof` no puede ser usada en una macro `#if` line, ya que el preprocesador no parsea tipos. Sin embargo, las expresiones en `#define` no son evaluadas por el preprocesador, así que el código es legal en este punto.

### 3.4. Apuntadores a Estructuras

Para ilustrar este punto, se presenta una versión del programa contador de palabras claves usando apuntadores en lugar de arreglos. La declaración externa de `keytab` no requiere ser cambiada, pero `main` y `binsearch` necesitan modificación.

```

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100

int getword(char *, int);
struct key *binsearch(char *, struct key *, int);
/* count C keywords; pointer version */

main()
{
    char word[MAXWORD];
    struct key *p;
    while (getword(word, MAXWORD) != EOF)

```

```

    if (isalpha(word[0]))
        if ((p=binsearch(word, keytab, NKEYS)) != NULL)
            p->count++;
    for (p = keytab; p < keytab + NKEYS; p++)
        if (p->count > 0)
            printf("%4d %s\n", p->count, p->word);
    return 0;
}

/* binsearch: find word in tab[0]...tab[n-1] */
struct key *binsearch(char *word, struct key *tab, int
    n)
{
    int cond;
    struct key *low = &tab[0];
    struct key *high = &tab[n];
    struct key *mid;
    while (low < high)
    {
        mid = low + (high-low) / 2;
        if ((cond = strcmp(word, mid->word)) < 0)
            high = mid;
        else if (cond > 0)
            low = mid + 1;
        else
            return mid;
    }
    return NULL;
}

```

Código 13: Keyword Count V2

Existen varios aspectos que vale la pena resaltar. Primero, la declaración de la función *binsearch* debe indicar que retorna un apuntador a una estructura en lugar de un entero; Si esta función encuentra la palabra, retorna el apuntador a ésta; si no la encuentra, retorna *NULL*.

En segundo lugar, los elementos de *keytab* deben ser accedidos por apuntadores. La inicialización para *low* y *high* son ahora apuntadores al comienzo y última posición de la tabla. El cambio más importante es ajustar el algoritmo para asegurarse que no genere un apuntador ilegal o intente acceder un elemento fuera del arreglo.

Finalmente, en el *main* se encuentra la instrucción:

```
for (p = keytab; p < keytab + NKEYS; p++),
```

Si `p` es un apuntador a una estructura, la aritmética sobre `p` toma en cuenta el tamaño de la estructura, así que `p++` incrementa `p` la cantidad correcta para moverse al próximo elemento a través del espacio de direcciones de memoria. Sin embargo, no se debe asumir que el tamaño de la estructura es la suma de sus miembros ya que en ocasiones se requiere un alineamiento de los objetos.

### 3.5. Typedef

C provee una facilidad llamada *typedef* para crear nuevos nombres para tipos de datos. Por ejemplo, la declaración:

```
typedef int length;
```

Hace que el nombre *length* sea un sinónimo para *int*. Posteriormente, este puede ser usado para otras declaraciones.

Otros ejemplos son:

```
Length len, maxlen;  
Length *lengths[];
```

De manera similar, la declaración `typedef char *String;`, hace que `String` sea un sinónimo para `char *` o apuntador a carácter, el cual puede ser usado en:

```
declarations and casts:  
String p, lineptr[MAXLINES], alloc(int);  
int strcmp(String, String);  
p = (String) malloc(100);
```

Se debe notar que el tipo declarado en `typedef` aparece en la posición del nombre de la variable, no después de la palabra `typedef`.

## 4. Entrada/Salida

La entrada y salida de un sistema operativo se refiere a la interacción de éste con un tercero. Ésta es posible mediante la entrada y salida de cadena de caracteres hacia y desde el sistema. Los periféricos de la computadora como el teclado y la pantalla son los dispositivos por defecto para esta tarea.

El tema tratado en este capítulo no es realmente parte del lenguaje C, sin embargo existe la necesidad de hacer énfasis. El estándar de C Ansi define este conjunto de librerías precisamente para que en cualquier sistema donde exista C pueda haber interacción. Se hará hincapié en las librerías `<stdio.h>`, `<string.h>` y `<ctype.h>`.

Estas librerías implementan un modelo simple para hacer entrada y salida de texto. Un flujo de texto consiste de una secuencia de líneas; cada línea termina con el carácter de nueva línea “`\n`”.

El mecanismo más simple de lectura de un carácter por vez de la entrada estándar, normalmente desde el teclado, es con la función `int getchar(void)`. Ésta retorna el próximo carácter leído cada vez que es llamado, o `EOF` si encuentra el final del archivo. El símbolo constante `EOF` (End of File) está definido en `<stdio.h>`.

Contrariamente, la función `int putchar(int)` es usada para colocar el entero indicado como parámetro en la salida estándar, que por defecto en la pantalla. Ésta función retorna el carácter escrito, o `EOF` si un error ocurre. Como se mostró en otros ejemplos la función `printf` realiza el mismo trabajo pero con un string como parámetro.

Para usar estas funciones se debe mostrar en el encabezado la instrucción: `#include <stdio.h>`. Por ejemplo, considere el programa siguiente que convierte la entrada en letras minúsculas:

```
#include <stdio.h>
#include <ctype.h>
main() /* lower: convert input to lower case */
{
    int c
    while ((c = getchar()) != EOF)
        putchar(tolower(c));
    return 0;
}
```

Código 14: Lower

La función `tolower` está definida en `<ctype.h>`; convierte un string de

mayúscula a minúscula.

## 4.1. Manejo de Salida y Entrada de Texto - Printf y Scanf

La función *printf* está definida por:

```
int printf(char *format, arg1, arg2, ...;
```

Esta función convierte, formatea e imprime sus argumentos a la salida estándar. Retorna el número de caracteres impresos.

El string dado en el parámetro "format" contiene dos tipos de objetos: caracteres ordinarios, que son copiados a la salida, y caracteres que especifican una conversión, esto causa conversión e impresión del argumento siguiente. Cada conversión comienza con el signo % y termina con la variable que se desea transformar. Los caracteres de conversión se especifican en la siguiente tabla expresando también su tipo y la manera en que será impreso:

- d,i; entero; número decimal
- o; entero; número octal sin signo
- x,X; entero; número hexadecimal sin signo
- u; entero; número decimal sin signo
- c; entero; caracter
- s; string (char \*); imprime la cadena de caracteres
- f; doble; [-]m.dddddd , donde el número de d's son dadas por la precisión (6 por defecto)
- e,E; doble; [-]m.dddddd e+/-xx o [-]m.ddddddE+/-xx, donde el número de d's cumplen lo anterior
- g,G; doble; usa %e o %E si el exponente es menor que -4 o mayor o igual que la precisión, sino usar %f.
- p; void \*; apuntador
- %; no convierte ningún argumento, imprime %

Además, se puede especificar la precisión con "\*", en cuyo caso el valor es calculado por la conversión del próximo argumento (debe ser un entero). Por ejemplo, para imprimir a lo sumo "max" caracteres de un string "s" se usa `printf("%.*s", max, s)`;

Algunas variaciones para la impresión de strings se dá a continuación:

```

:%s: :hello, world:
:%10s: :hello, world:
:%.10s: :hello, wor:
:%-10s: :hello, world:
:%.15s: :hello, world:
:%-15s: :hello, world :
:%15.10s: : hello, wor:
:%-15.10s: :hello, wor :

```

La función *sprintf* realiza la misma conversión de *printf*, pero guarda la salida en un string:

```
int sprintf(char *string, char *format, arg1, arg2, ...);
```

Esta función revisa los argumentos en *arg1*, *arg2*, etc., de acuerdo al formato explicado previamente, pero coloca el resultado en "string" en lugar de la salida estándar.

Un buen ejemplo es mostrado a continuación, que muestra una versión mínima de *printf*, que procesa una lista de longitud variable de argumentos:

```

#include <stdarg.h>

/* minprintf: minimal printf with variable argument
   list */

void minprintf(char *fmt, ...)
{
    va_list ap; /* points to each unnamed arg in turn */
    char *p, *sval;
    int ival;
    double dval;
    va_start(ap, fmt); /* make ap point to 1st unnamed
        arg */
    for (p = fmt; *p; p++)
    {
        if (*p != '%')
        {
            putchar(*p);
            continue;
        }
        switch (*++p)
        {
            case 'd':
                ival = va_arg(ap, int);

```

```

        printf("%d", ival);
        break;
    case 'f':
        dval = va_arg(ap, double);
        printf("%f", dval);
        break;
    case 's':
        for (sval = va_arg(ap, char *); *sval; sval++)
            putchar(*sval);
        break;
    default:
        putchar(*p);
        break;
    }
}
va_end(ap); /* clean up when done */
}

```

Código 15: Versión Mínima de *printf*

Contrariamente, la función *scanf* que es análoga a *printf*, *int scanf(char \*format, ...)*, lee caracteres de la entrada estándar, los interpreta de acuerdo al formato especificado en *format*, y guarda el resultado en los sucesivos argumentos; el resto de los argumentos, debe ser un apuntador que indica donde el correspondiente string debe ser almacenado.

Existe una función similar, *sscanf* que lee de un string en lugar de la entrada estándar, *int sscanf(char \*string, char \*format, arg1, arg2, ...)*. Revisa el string de acuerdo al formato en "format" y guarda el resultado en los sucesivos argumentos, éstos deben ser apuntadores.

La especificación de los formatos viene dada por la tabla mostrada previamente.

## 4.2. Manejo de Archivos

Los ejemplos vistos previamente solo tomaban en cuenta lectura y escritura de texto por la salida y entrada estándar. El próximo paso es escribir un programa que acceda archivos externos. Un programa que ilustra la necesidad para tal operación es el comando de UNIX *cat*, que concatena un conjunto de archivos en la salida estandar. Éste es usado también para imprimir archivos en la pantalla. Por ejemplo, el comando:

```
cat x.c y.c
```

Imprime el contenido de los archivos `x.c` y `y.c` en la salida estándar. El asunto es cómo implementar esto, es decir, cómo arreglar los archivos de manera que cada nombre de archivo sea pasado como argumento y sea leído. Aquí se mostrará cómo leer los datos de archivos externos.

Las reglas son sencillas, antes de que un archivo sea leído o escrito, debe ser abierto por la función `fopen` de la librería `<stdio.h>`. Ésta función toma un nombre como `x.c` o `y.c`, realiza una transacción con el sistema operativo y retorna un apuntador que será usado para las posteriores lecturas y escrituras.

Este apuntador llamado *apuntador a un archivo*, apunta a una estructura definida como `FILE` que contiene información sobre el archivo como la ubicación de un buffer, la posición al caracter actual en el buffer, si el archivo está siendo leído o escrito, y si ha ocurrido un error o EOF. Sin embargo, los usuarios no necesitan saber estos detalles.

Para declarar un apuntador de este tipo se utiliza por ejemplo: `FILE *fp;`, esto dice que `fp` es un apuntador a un archivo. La llamada para abrir un archivo en un programa es:

```
fp = fopen(name, mode);
```

El primer argumento de `fopen` es un string que contiene el nombre del archivo, el segundo es el modo de apertura del archivo: lectura ("`r`"), escritura ("`w`") y agregar ("`a`"). Si un archivo que no existe es abierto para escritura o para añadir, es creado si es posible. Abrir archivos existente para escritura causa que el viejo contenido sea desechado, mientras que se indica el modo "append" preserva el contenido.

El abrir un archivo que no existe para lectura indicará un error. Otras causas de error son tratar de leer un archivo cuando no tiene los permisos apropiados. Si existe algún error, `fopen` retornará `NULL`.

Existen un par de funciones útiles que permitirán la escritura y lectura de un archivo, una vez abierto. `getc` retorna el próximo caracter de un archivo a ser leído, EOF si encuentra final de archivo o error: `int getc(FILE *fp)`

Por otro lado, `putc` es una función para escribir en un archivo:

```
int putc(int c, FILE *fp)
```

`putc` escribe el caracter `c` al archivo `fp` y retorna el caracter escrito, o EOF si un error ocurrió.

Cuando un programa en C se ejecuta, el sistema operativo es responsable de abrir tres archivos y provee apuntadores para éstos. Estos archivos son la entrada estándar, la salida estándar y el error estándar; los apuntadores correspondientes son denominados `stdin`, `stdout`, y `stderr`, y son declarados en `<stdio.h>`. Normalmente, `stdin` esta asociado al teclado, y `stdout` y `stderr` están asociados a la pantalla. Sin embargo, `stdin` y `stdout` pueden ser redirigidos a otros archivos.

Además, esta librería ofrece las funciones `fscanf` y `fprintf` para escritura

y lectura de un archivo. Éstas son idénticas a *scanf* y *printf*, excepto que el primer argumento es un apuntador a archivo que especifica el archivo a ser leído y escrito.

```
int fscanf(FILE *fp, char *format, ...)
int fprintf(FILE *fp, char *format, ...)
```

Con este conocimiento, estamos en la capacidad de interpretar un programa similar al *cat* de UNIX que usa las funciones explicadas previamente

```
#include <stdio.h>

/* cat: concatena archivos */
main(int argc, char *argv[])
{
    FILE *fp;
    void filecopy(FILE *, FILE *)
    if (argc == 1) /* no args; pide de stdin */
        filecopy(stdin, stdout);
    else
        while(--argc > 0)
            if ((fp = fopen(++argv, "r")) == NULL)
            {
                printf("cat: can't open %s\n", *argv);
                return 1;
            }
            else
            {
                filecopy(fp, stdout);
                fclose(fp);
            }
            return 0;
}

/* filecopy: copia archivo ifp a ofp */
void filecopy(FILE *ifp, FILE *ofp)
{
    int c;
    while ((c = getc(ifp)) != EOF)
        putc(c, ofp);
}
```

Código 16: Versión propia de *cat*

Los apuntadores `stdin` y `stdout` son objetos de tipo `FILE *`. Éstos son constantes, por esto no es posible asignarles algo.

Finalmente, la función `fclose` realiza lo inverso de `fopen`, cierra la conexión entre el apuntador al archivo y el propio archivo. Por algunos parámetros que se establecen en el sistema operativo, es buena idea ejecutar esta operación siempre que se abre un archivo.

```
int fclose(FILE *fp)
```

A continuación se presenta un implementación de `cat` más segura:

```
#include <stdio.h>

/* cat: concatena archivos , version 2 */
main(int argc, char *argv[])
{
    FILE *fp;
    void filecopy(FILE *, FILE *);
    char *prog = argv[0]; /* nombre del programa para
        errores */
    if (argc == 1) /* no args; copia de stdin */
        filecopy(stdin, stdout);
    else
        while (--argc > 0)
            if ((fp = fopen(++argv, "r")) == NULL)
            {
                fprintf(stderr, "%s: can't open %s\n", prog, *
                    argv);
                exit(1);
            }
            else
            {
                filecopy(fp, stdout);
                fclose(fp);
            }
            if (ferror(stdout))
            {
                fprintf(stderr, "%s: error writing stdout\n",
                    prog);
                exit(2);
            }
        }
    exit(0);
}
```

---

### Código 17: Versión más segura de *cat*

En este ejemplo se puede ilustrar el uso del apuntador *stderr*

Este programa chequea errores de dos maneras. Primero, la salida de diagnóstico producida por *fprintf* se redirige a *stderr*. Segundo, el programa usa la función *exit*, que termina la ejecución del programa cuando es llamado. El argumento de esta función es el que se retorna a cualquiera que llame a la función, puede indicar estatus de éxito o error. De forma convencional, un valor de retorno cero indica que no ocurrió algún error, y distinto de cero indica una situación de falla del programa. Esta es similar a utilizar `return expr`.

La función *ferror* retorna distinto cero si un error ocurrió en el archivo apuntado por *fp*.

```
int ferror(FILE *fp)
```

### 4.3. Funciones Misceláneas

- `char *fgets(char *line, int maxline, FILE *fp)`; esta función lee la próxima línea a ser leída de un archivo y la retorna. Retorna NULL si llego a EOF u ocurrió un error.
- `int fputs(char *line, FILE *fp)`; retorna EOF si ocurrió un error, y un valor no negativo en caso contrario. La librería de funciones *gets* y *puts* son similares a *fgets* y *fputs*, pero operan sobre *stdin* y *stdout*.

A continuación se listan las funciones más usadas de la librería `<string.h>`:

- `strcat(s,t)`, concatena *t* al final de *s*.
- `strncat(s,t,n)`, concatena *n* caracteres de *t* al final de *s*.
- `strcmp(s,t)` retorna negativo, cero, o positivo para los casos  $s < t$ ,  $s == t$ ,  $s > t$  respectivamente.
- `strncmp(s,t,n)` igual que `strcmp` pero solo en los primeros *n* caracteres.
- `strcpy(s,t)` copia *t* en *s*.
- `strncpy(s,t,n)` copia a lo sumo *n* caracteres de *t* en *s*
- `strlen(s)` retorna la longitud de *s*
- `strchr(s,c)` retorna el apuntador al primer carácter *c* en *s*, o NULL si no está presente.

- `strrchr(s,c)` retorna un apuntador al último caracter `c` en `s`, o `NULL` si no esta presente.

Algunas funciones comunes de la libreria `<ctype.h>` que realizan conversiones y verificación de caracteres.

- `isalpha(c)` distinto de cero si `c` es un caracter del alfabeto, 0 si no.
- `isupper(c)` distinto de cero si `c` esta en mayúscula, 0 si no.
- `islower(c)` distinto de cero si `c` esta en minúscula, 0 si no.
- `isdigit(c)` distinto de cero si `c` es un dígito, 0 si no.
- `isalnum(c)` distinto de cero si `isalpha(c)` o `isdigit(c)`, 0 si no
- `isspace(c)` distinto de cero si `c` es un espacio en blanco, tabulador, nueva línea, entre otros.
- `toupper(c)` retorna `c` convertida a mayúscula.
- `tolower(c)` retorna `c` convertido a minúscula.

Es importante conocer algunas funciones de uso avanzado como:

- Para ejecución de comandos la función `system(char *s)` ejecuta el comando contenido en el string `s`. Por ejemplo, `system("date")`.
- Para reserva de espacio de memoria para estructuras, las funciones `malloc` y `calloc` obtienen los bloques de memoria dinámicamente, los prototipos son: `void *malloc(size_t n)`, que retorna un apuntados a `n` bytes de almacenamiento, o `NULL` si la petición no se puede cumplir; `void *calloc(size_t n, size_t size)`, retorna un apuntador a suficiente espacio de memoria para un arreglo de `n` objetos del tipo especificado, o `NULL` si no se puede cumplir la petición. Por ejemplo:  
`ip = (int *) calloc(n, sizeof(int));`
- `free(p)`, libera el espacio señalado por el apuntador en `p`. La manera mas segura de liberar espacio es:

```
for (p = head; p != NULL; p = p->next) /* INCORRECTO */
    free(p);

for (p = head; p != NULL; p = q) /* CORRECTO */
{
```

```
q = p->next;  
free(p);  
}
```

Algunas funciones matemáticas se encuentran disponibles en `<math.h>`

- `sin(x)` sine of x, x in radians
- `cos(x)` cosine of x, x in radians
- `atan2(y,x)` arctangent of y/x, in radians
- `exp(x)` exponential function  $e^x$
- `log(x)` natural (base e) logarithm of x ( $x > 0$ )
- `log10(x)` common (base 10) logarithm of x ( $x > 0$ )
- `pow(x,y)`  $x^y$
- `sqrt(x)` square root of x ( $x > 0$ )
- `fabs(x)` absolute value of x

## 5. Funciones

Las funciones dividen grandes tareas de cómputo en varias más pequeñas, y permiten la posibilidad de construir sobre lo que otros ya han hecho, es decir, permite la reutilización de código. Las funciones apropiadas ocultan los detalles de operación de las partes de un programa, así dan claridad a la totalidad y facilitan la penosa tarea de hacer el mismo cambio numerosas veces.

Este lenguaje se diseñó para hacer que las funciones fueran eficientes y fáciles de usar; los programas escritos en C se componen de muchas funciones pequeñas en lugar de sólo algunas grandes. Un programa puede estar compuesto de uno o más archivos de código. Ésto es posible ya que las fuentes o archivos de código pueden ser compilados separadamente para ser posteriormente enlazados entre ellos e incluso con librerías compiladas previamente del sistema.

Una función en C debe ser declarada y definida, en el primer caso, se le conoce como prototipo de la función donde se indica el tipo de retorno, el nombre y los argumentos de la función. Posteriormente, una función es definida indicando la información del prototipo nuevamente y el cuerpo de la función, es decir, las líneas de código que ejecuta.

Un ejemplo clásico, es una función que lea cada línea de la entrada estándar o un archivo y verifique que contenga un patrón particular. Puede observar en su terminal de comandos de Unix, que cumple la misma funcionalidad del comando *grep*.

La forma óptima de desarrollar este programa es escribir tres funciones que distribuyan la tarea, sin embargo, siempre se pueden escribir todas las instrucciones en la función principal sin el uso de subrutinas auxiliares. Esta función posee el nombre predeterminado de *main* y es la que representa el conjunto de instrucciones principales que se ejecutan por defecto.

El código se presenta a continuación:

```
#include <stdio.h>
#define MAXLINE 1000 /* longitud maxima de la lines */

/* Declaracion de funciones */
int getline(char line [], int max)
int strindex(char source [], char searchfor []);

char pattern [] = "ould"; /* patron a buscar */

/* Función principal */
```

```

main() {
    char line[MAXLINE];
    int found = 0;
    while (getline(line, MAXLINE) > 0)
        if (strindex(line, pattern) >= 0)
        {
            printf("%s", line);
            found++;
        }
    return found;
}

/* getline: guarda la linea en la variable s
y retorna la longitud */
int getline(char s[], int lim) {
    int c, i;
    i = 0;
    while (--lim > 0
        && (c=getchar()) != EOF && c != '\n')
        s[i++] = c;
    if (c == '\n')
        s[i++] = c;
    s[i] = '\0';
    return i;
}

/* strindex: retorna el indice de t en s,
si no -1 */
int strindex(char s[], char t[]) {
    int i, j, k;
    for (i = 0; s[i] != '\0'; i++) {
        for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
            ;
        if (k > 0 && t[k] == '\0')
            return i;
    }
    return -1;
}

```

Código 18: Grep propio

El pseudocódigo a muy alto nivel para mayor claridad es:

```

while (no existen mas lineas)
    if (la linea contiene el patron)
        imprimir la linea

```

La función que obtiene las líneas y a su vez, verifica que existan más líneas, es *getline*. La función para imprimir la línea, es la conocida *printf* contenida en la librería *stdio.h* incluida en el encabezado del archivo. La que revisa si el patron se encuentra en la cadena de caracteres es *strindex(s,t)*, ésta retorna la posición o índice del primer caracter t que consigue de izquierda a derecha en el string s. El valor negativo de retorno indica que no se encuentra el caracter en la línea especificada.

De manera general, la definición de una función posee la siguiente estructura:

```

tipo-retorno nombre-funcion(declaracion argumentos)
{
    declaraciones y sentencias
}

```

Dado el caso en que no se interese retornar algún valor, el tipo de retorno puede ser especificado como *void*. En caso contrario, se debe especificar al final de la función una última instrucción *return expresión*. Esta expresión será convertida en el tipo declarado en el prototipo de la función de ser necesario. Además, es opcional que la función que realice la llamada capture en una variable el valor de retorno.

La mecánica de la carga y compilación de un programa en C varia dependiendo del sistema. En el caso de los sistemas Unix, utilizados para este curso, se utiliza el comando *gcc*. Por ejemplo, suponiendo que las tres funciones están guardadas en tres archivos separados *main.c*, *getline.c* y *strindex.c*; para compilarlos se ejecuta en un shell de comandos:

```
gcc main.c getline.c strindex.c
```

Esto compilará los tres archivos, generando un archivo con el código objeto o máquina con extensión ".o" asociado a cada uno: *main.o*, *getline.o* y *strindex.o*. Luego, serán cargados en un archivo ejecutable *a.out*. En caso de errores de compilación, serán indicados en la salida estándar.

Un par de ejemplos de funciones que retornan tipos más complejos que vacío (void) o números enteros (int) se presentan a continuación:

```

#include <ctype.h>
/* atof: convierte un string s
   en un punto flotante de precisión doble */

```

```

double atof(char s[]) {
    double val, power;
    int i, sign;
    /* salta espacios en blanco */
    for (i = 0; isspace(s[i]); i++)
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')
        i++;
    for (val = 0.0; isdigit(s[i]); i++)
        val = 10.0 * val + (s[i] - '0');
    if (s[i] == '.')
        i++;
    for (power = 1.0; isdigit(s[i]); i++)
    {
        val = 10.0 * val + (s[i] - '0');
        power *= 10;
    }
    return sign * val / power;
}

```

Código 19: atof propio

Una función similar se encuentra en una librería estándar del paquete de instalación del compilador de C, llamada *stdlib.h*. Como se nota en la declaración de la función, ésta retorna un valor de tipo *double*. Cualquier programa que la utilice debe conocer que retorna un valor distinto a un entero.

Un programa que puede hacer uso de la rutina *atof* es el de una calculadora rudimentaria:

```

#include <stdio.h>
#define MAXLINE 100

main() {
    double sum, atof(char []);
    char line [MAXLINE];
    int getline(char line [], int max);
    sum = 0;
    while (getline(line, MAXLINE) > 0)
        printf("\t\t%g\n", sum += atof(line));
    return 0;
}

```

```
}
```

## Código 20: Calculadora rudimentaria

La declaración `double sum, atof(char [])`; indica que la variable *sum* es de tipo *double* y que la función *atof(char [])* toma un argumento de tipo *char[]* y retorna un argumento de tipo *double*. Ésta última debe ser declarada y definida consistentemente. Esto es, si en el prototipo de la función se indica que retorna o toma parámetros de algún tipo en particular, la definición de la función debe indicar lo mismo, si no dará un error a nivel de compilación.

La declaración y definición puede parecer redundante, pero este es un estándar utilizado por todos los programadores de este lenguaje. Esto permite reutilización e inclusión eficiente de funciones. En caso de no declarar un prototipo, se asume que la primera aparición de la función es el prototipo.

Dada la función *atof* correctamente declarada, se puede definir las *atoi* de la siguiente manera:

```
int atoi(char s[])
{
    double atof(char s[]);
    return (int) atof(s);
}
```

La expresión `return expresion`, indica que lo retornado por *atof(s)* es convertido en un tipo entero justo antes de terminar de ejecutar. En esta instrucción se da la conversión de *double* a *int*, esta operación descarta parte de la información por lo explicado en previas secciones, pero colocar explícitamente el cast elimina advertencias a nivel de compilación.

## 6. Estructura de un programa

En general, un programa consta de un cierto número de funciones y una función *main*. Es usual que estas funciones sean agrupadas de acuerdo a un aspecto en común, y así mismo distribuidas entre distintos archivos fuente identificados con un nombre nemónico que permita asumir que tipo de funciones se pueden encontrar allí.

De la misma manera, es un estándar que cada fuente tenga un archivo extensión ".h" y ".c" asociado. Donde en el primer caso, se incluyen la declaración de variables, estructuras y funciones globales. En el segundo caso, se coloca el código asociado a cada función global y otras variables o funciones locales necesarias. También es común en el caso de que aplique, se tenga un

archivo con el código de la función principal *main()*, aunque para ésta no se especifica el prototipo.

## 6.1. Variables externas

Un programa en C consiste de un conjunto de objetos externos, que pueden ser variables o funciones. Se pueden definir variables externas, como aquellas declaradas afuera del cuerpo de una función y que se encuentran disponibles para cualquiera. En el caso de las funciones, siempre son externas, ya que C no permite que otras funciones sean definidas dentro de otras.

Cualquier función puede acceder una variable externa haciendo referencia por su nombre, si éste ha sido declarado en algún punto. Si una cantidad grande de variables debe ser compartida entre varias funciones, usar variables externas es la manera más conveniente y eficiente de hacerlo. Sin embargo, se debe ser precavido en hacerlo solo si es necesario, ya que si no, quedan muchas referencias inútiles entre funciones.

Estas variables son ideales debido a su alcance y tiempo de vida. Esto es, automáticamente, las variables declaradas en una función son internas, ellas entran en existencia cuando se empieza la ejecución de la función y desaparecen cuando acaba. Por otro lado, las variables externas son de existencia permanente.

Una implementación del programa de la calculadora es mostrada a continuación, donde los operadores usan una notación infija (i.e.  $(1-2)*(4+5) = 1\ 2\ -\ 4\ 5\ +\ *$ ). Cada operando es empilado y cuando es un operador, dependiendo si es unario o binario, la cantidad de operandos apropiados son sacados de la pila para aplicar la operación, este resultado es empilado. La estructura del programa es un ciclo que realiza la operación apropiada en cada operador y operando al momento que se procesan.

```
while (existan operadores u operandos)
  if(numero)
    empilar;
  else if(operador)
    desempilar operandos
    aplicar operacion
    empilar resultado
  else if(nueva linea)
    desempilar tope de la pila e imprimir
  else
    error
```

La cuestión es decidir como manejar la pila. Una posibilidad es declararla en el main, y pasarla como parámetro a las funciones de empilado (push) y desempilado (pop) junto con la posición actual del último elemento accedido. Pero main no necesita saber sobre las variables que controlan la pila, así que es mejor que éstas sean variables externas accesibles por las funciones *pop* y *push*. El código en C se muestra a continuación:

```
#include <stdio.h>
#include <stdlib.h> /* para atof() */

/* tamaño máximo de
un operando u operador */
#define MAXOP 100
/* señal de que un
numero fue encontrado */
#define NUMBER '0'

int getop(char []);
void push(double);
double pop(void);

/* reverse Polish calculator */
main()
{
    int type;
    double op2;
    char s[MAXOP];
    while ((type = getop(s)) != EOF)
    {
        switch (type)
        {
            case NUMBER:
                push(atof(s));
                break;
            case '+':
                push(pop() + pop());
                break;
            case '*':
                push(pop() * pop());
                break;
            case '-':
```

```

        op2 = pop();
        push(pop() - op2);
        break;
    case '/':
        op2 = pop();
        if (op2 != 0.0)
            push(pop() / op2);
        else
            printf("error: zero divisor\n");
        break;
    case '\n':
        printf("\t%.8g\n", pop());
        break;
    default:
        printf("error: unknown command %s\n", s);
        break;
    }
}
return 0;
}

```

Código 21: Calculadora Polaca Reversa

Es importante destacar que en el caso de los operadores de resta y división el orden debe ser distinguido, es decir, es incorrecta la operación `push(pop() - pop())` por esto se utiliza una variable temporal. Las operaciones de pila se definen a continuación:

```

#define MAXVAL 100
/* valor maximo de profundidad de la pila */

int sp = 0; /* proxima posicion libre de la pila */
double val[MAXVAL]; /* pila */

/* push: empila f */
void push(double f)
{
    if (sp < MAXVAL)
        val[sp++] = f;
    else
        printf("error: stack full, can't push %g\n", f);
}

```

```

/* pop: desempila y retorna el valor del top */
double pop(void)
{
    if (sp > 0)
        return val[--sp];
    else
    {
        printf("error: stack empty\n");
        return 0.0;
    }
}

```

Código 22: Empilado y Desempilado

De la misma manera, la función *getop* obtiene el próximo operador u operando. La tarea es facil, solo debe ignorar los espacios en blanco y de tabulación. Además, si el carácter no es un dígito, retornarlo. Y por otro lado, coleccionar una cadena de dígitos y retornar NUMBER, la señal de que un número ha sido encontrado.

```

#include <ctype.h>

int getch(void);
void ungetch(int);

/* getop: obtiene el proximo
caracter u operando numerico */
int getop(char s[])
{
    int i, c;
    while ((s[0] = c = getch()) == ' ' || c == '\t')
        ;
    s[1] = '\0';
    if (!isdigit(c) && c != '.')
        return c; /* no es un numero */
    i = 0;
    if (isdigit(c)) /* guardar los enteros */
        while (isdigit(s[++i] = c = getch()))
            ;
    if (c == '.') /* caso de los reales */
        while (isdigit(s[++i] = c = getch()))
            ;
}

```

```

s[i] = '\0';
if (c != EOF)
    ungetch(c);
return NUMBER;
}

```

Código 23: Lectura de operandos y operadores

```

#define BUFSIZE 100
char buf[BUFSIZE]; /* buffer */
int bufp = 0; /* proxima posicion libre en buf */

/* obtiene un caracter */
int getch(void)
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

/* coloca un caracter en el input */
void ungetch(int c)
{
    if (bufp >= BUFSIZE)
        printf("ungetch: too many characters\n");
    else
        buf[bufp++] = c;
}

```

Código 24: Lectura de caracteres

En cuanto a las funciones *getch* y *ungetch* también se incluyen en una librería estándar de C.

## 6.2. Archivos de Encabezado

Siguiendo el ejemplo de la calculadora, vamos a dividir el programa en distintos archivos fuentes. La función *main* se coloca en un archivo *main.c*; las funciones *push* y *pop* y sus variables se ubican en el archivo *stack.c*; *getop* se coloca en *getop.c*. Finalmente, *getch* y *ungetch* se ubican en *getch.c*.

Existe algún detalle más por tomar en cuenta, estas son las definiciones y declaraciones compartidas entre los archivos. De esta manera, se colocarán estas instrucciones comunes en un archivo de encabezado, *calc.h*, resultando en:

```

#define NUMBER '0'

```

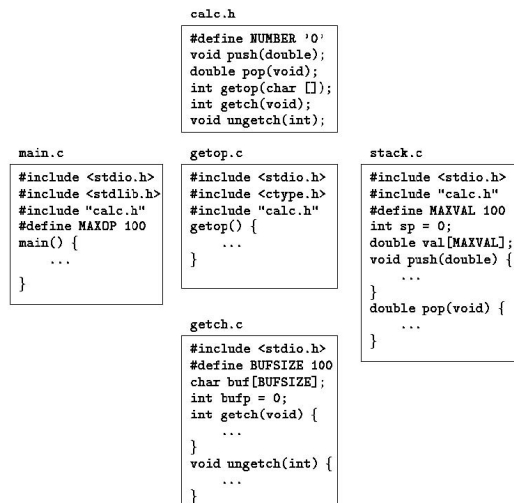
```

void push(double);
double pop(void);
int getop(char []);
int getch(void);
void ungetch(int);

```

Código 25: Lectura de caracteres

Un vistazo general de la estructura del código sería:



Para este programa se pensó dado su tamaño que solo era necesario un archivo de encabezado, sin embargo, para programas más extensos se aconseja tener tantos archivos de encabezado como lo requiera la estructuración del código.

### 6.3. Estructura de Bloques

C no es un lenguaje con estructura de bloques en el sentido de Pascal u otros lenguajes similares, ya que las funciones no pueden ser definidas en otras funciones. Por otro lado, las variables pueden ser definidas en una estructura de bloque. Un bloque es un conjunto de instrucciones encerradas entre llaves.

Por ejemplo, las variables declaradas en un bloque sólo existen dentro de él.

```

if (n > 0)
{
    int i;
    for (i = 0; i<n; i++)
        ...
}

```

El alcance de la variable  $i$  se limita al bloque; esa  $i$  no esta relacionada con alguna otra  $i$  fuera él. Una variable declarada e inicializada en un bloque se inicializa cada vez que se ejecuta, por esto no es recomendable utilizar este estilo de programación al menos que sea la intención.

Por ejemplo,

```

int x;
int y;

f(double x)
{
    double y;
}

```

En este caso, las ocurrencias de la variable  $x$  en el bloque no se refiere al entero, si no al doble. Afuera de la función  $f$ , se refiere al entero.

## 6.4. Inicialización

La inicialización consiste en darle un primer valor a una variable, en la ausencia de una inicialización explícita, las variables externas y estáticas se le da un valor de cero por defecto.

Las variables escalares pueden ser inicializadas al ser definidas, seguida de un signo de igualdad y una expresión:

```

int x = 1;
char squota = '\';
long day = 1000L * 60L * 60L * 24L;
int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

```

## 6.5. Recursión

Una función recursiva posee en su cuerpo llamadas directas o indirectas a sí misma. Un clásico ejemplo de recursión es el algoritmo de **Quicksort** desarrollado por C.A.R. Hoare en 1962.

Dado un arreglo, un elemento es elegido y el resto es particionado en dos subconjuntos, en aquellos menores y mayores que el elemento pivote. El mismo proceso es luego aplicado recursivamente a los dos subconjuntos. Cuando uno de los subconjuntos tiene menos de dos elementos, ya no requiere ordenamiento y se detiene la recursión.

La versión de Quicksort que se presenta no es la más rápida, pero es la más simple, ya que se usa el elemento medio para hacer el particionamiento.

```

/* qsort: sort v[left]...v[right] en orden creciente */
void qsort(int v[], int left, int right)
{
    int i, last;
    void swap(int v[], int i, int j);

    if (left >= right)
        return; /* menos de dos elementos */

    swap(v, left, (left + right)/2);
    last = left; /* a v[0] */
    for (i = left + 1; i <= right; i++) /* partición */
        if (v[i] < v[left])
            swap(v, ++last, i);
    swap(v, left, last); /* restaura elem */
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}

/* swap: intercambia v[i] y v[j] */
void swap(int v[], int i, int j)
{
    int temp;
    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}

```

Código 26: Quicksort

La librería estándar de C incluye una versión *qsort* que ordena objetos de cualquier tipo. El código recursivo es más compacto pero como mantiene una pila necesita más memoria para su ejecución.

## 6.6. El preprocesador de C

Este es el primer paso de la compilación, contiene un lenguaje peculiar distinto al usado en un programa en C regularmente, cada expresión es denominada *macro*. Algunas características útiles son presentadas en esta sección.

### 6.6.1. Inclusión de Archivos

Esta directiva indica qué archivo se desea incluir al archivo fuente, existen dos formas en que puede ser dada `#include 'archivo'` o `#include <archivo>`. Si el parámetro archivo se encuentra entre comillas dobles, indica que este archivo se encuentra en una ruta de archivos inusual o elegida por el usuario. En el otro caso, indica que el archivo se encuentra en una directorio conocido por el compilador, en general el directorio de instalación.

Usualmente se encuentran numerosas líneas de inclusión al inicio de un archivo fuente, para incluir directivas de tipo `#define` y declaraciones de tipo `extern`, o para acceder prototipos de funciones de las librerías más comunes como `<stdio.h>`. Esta directiva es útil ya que permite unir declaraciones de funciones en el caso de un código extenso y le dá entendimiento al programa.

### 6.6.2. Substituciones Macro

Una definición tiene la forma

```
#define nombre texto de reemplazo
```

Este es la variación más simple de este tipo de macros. Esto indica que todas las ocurrencias del string "nombre" será reemplazado por "texto de reemplazo". En general, "nombre" tiene la forma de una variable regular y el "texto de reemplazo" es arbitrario. Normalmente éste toma el valor hasta cuando acaba la línea, sin embargo, se puede separar en líneas siempre y cuando se coloque entre cada línea el carácter `^`. De manera estándar, éstas son declaradas en mayúscula.

Por ejemplo, si se define el string YES, no se sustituye en los strings entre doble comillas y substrings como `printf("YES")` o `YESMAN`.

Cualquier string y texto de reemplazo puede ser usado, por ejemplo:

```
#define forever for (;;) 
```

También es posible definir macros con argumentos, de manera que el texto de reemplazo sea distinto de acuerdo al argumento:

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

Si se asigna lo siguiente:

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

Será reemplazada por la línea

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

Si se examina el macro, se puede notar que ambas expresiones son evaluadas dos veces, por ejemplo la llamada `max(i++, j++)` es errónea ya esto incrementará la variable `i` e `j` en el doble.

De manera contraria, los nombres pueden ser indefinidos cuando sea requerido con la directiva `#undef`

### 6.6.3. Inclusión Condicional

Es posible controlar el preprocesamiento con un condicional que son evaluados durante esta etapa. Esto provee un manera de incluir código selectivamente, dependiendo del valor de la condición evaluada durante la compilación.

La directiva `#if` evalúa una expresión que resulta en un entero. Si la expresión es diferente de cero, la líneas siguientes hasta un `#endif` o `#elif` o `#else` son incluidas. La expresión `defined(nombre)` en `#if` es 1 si el nombre no ha sido definido, en caso contrario, es cero.

Por ejemplo, para asegurarse que el contenido del archivo `hdr.h` sea incluida exactamente una vez, el contenido de todo el archivo debe rodearse de un condicional así:

```
#if !defined(HDR)
#define HDR
...
#endif
```

Durante el preprocesamiento, el primer archivo que haga la inclusión de `hdr.h` define la variable `HDR`; las siguientes inclusiones que hagan otros archivos conseguirán que esta variable se encuentra definida, así que saltará la parte del código hasta `#endif`. Esto para evitar incluir un archivo muchas veces. Si este estilo se sigue de manera consistente, el programador no debe estar preocupado en cuántas veces incluyó un archivo o no.

La siguiente secuencia chequea `SYSTEM` y decide que versión del encabezado incluir:

```
#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
```

De la misma manera, existen dos directivas `#ifdef` y `ifndef` que son aún mas específicas dado que verifican si un nombre ya esta definido. Otra manera de realizar alguno de los ejemplos previos puede ser:

```
#ifndef HDR  
    #define HDR  
    ...  
#endif
```

## 7. Apuntadores

### 7.1. ¿Qué es un apuntador?

El lenguaje de programación C está repleto de apuntadores. Es realmente imposible dominar este lenguaje sin entender completamente qué son los apuntadores.

El problema de los apuntadores es que la mayoría de la gente que programa no tiene un conocimiento básico de cómo se manejan las variables en el lenguaje C. Una variable es básicamente un contenedor declarada de un tipo (int, short, long, etc) dónde se puede almacenar información, que es variable en el tiempo. Esto se dá porque el lenguaje C funciona en un mundo intermedio, entre la programación de alto nivel y el mundo de los bits.

Cuando se declara una variable en C, se le asigna un tipo y un nombre. Por debajo el compilador reserva un espacio en memoria del tamaño de la variable deseada (ver Código 2, página 9) . Si no se le asigna un valor a la variable inmediatamente, el espacio sigue reservado (Una razón muy fuerte para no declarar variables que no se usan). Una vez asignado un valor, el espacio en memoria mantiene esta información en el formato y con los límites necesarios.

Si se considera a una variable regular de C, que tiene un espacio en memoria donde se guarda el contenido y una dirección de memoria que indica **dónde** está la información, un apuntador es un tipo de variable que contiene una dirección de memoria para llegar a un contenido.

Todos los tipos de C tienen apuntadores y existen diversas formas de declarar estas variables, todas reconocidas por el compilador de C:

```
int* ptr;  
int *ptr;  
int * ptr;
```

Para mantener la consistencia, al declarar la variable apuntador se usará en esta guía la siguiente:

```
int *ptr;
```

En C, las variables declaradas sin asignación inmediata son inicializadas por el compilador a un valor predeterminado. Para los tipos numéricos (int, float, double) el valor es la representación del 0. Para todas las variables tipo apuntador este valor es **NULL**.

Una vez declarada una variable de tipo apuntador no es necesario usar **\*** para la asignación, ya que después de la asignación **ptr** es siempre una

variable tipo **apuntador a algo**. Si se usa **\*** antes de una variable esto tiene un significado completamente diferente, que se describirá más adelante.

Existen dos operadores unarios importantes al hablar de apuntadores. Uno es **&** y el otro es la misma **\*** usada para declarar variables de tipo apuntador.

Ya hemos dicho que toda variable en C tiene reservado un pedazo de la memoria existente. Tomando eso en cuenta tiene que haber una forma de obtener la dirección de la memoria donde está almacenado una información. El operador **&** hace justamente esto. Unido a una variable cualquiera, usando este operador, se obtiene la dirección de memoria donde está la información. Un ejemplo es la siguiente línea de código:

```
(...)  
int k = 6;  
printf("La direccion de k es: %p\n", &k);  
(...)
```

Una vez que se tiene una variable tipo apuntador, el agregarle una **\*** durante su asignación o utilizarla dentro del código, lo “deferencia”. Básicamente **\*ptr** afecta el contenido del espacio de memoria donde apunta ptr. Un ejemplo pequeño se muestra abajo:

```
(...)  
int *ptr;  
int k;  
ptr = &k;  
*ptr = 8;  
(...)
```

Para concretar estas ideas, mostramos el siguiente ejemplo:

```
1 #include<stdio.h>  
2  
3 int j, k, m;  
4 int *ptr;  
5  
6 int main(void) {  
7  
8     j = 1;  
9     k = 2;  
10    m = 3;  
11    ptr = &k;
```

```

12     k = m;
13     printf("\n");
14     printf("j tiene el valor %d y esta almacenado en %
        p\n", j , (void *)&j);
15     printf("k tiene el valor %d y esta almacenado en %
        p\n", k , (void *)&k);
16     printf("m tiene el valor %d y esta almacenado en %
        p\n", m , (void *)&m);
17     printf("ptr tiene el valor %p y esta almacenado en
        %p\n", ptr , (void
18     *)&ptr);
19     printf("El valor del entero apuntado por ptr es %d
        \n", *ptr);
20
21     return 0;
22 }

```

Código 27: Ejemplo de apuntadores

En este ejemplo tenemos las declaraciones de las variables **j**, **k** y **m**. En las líneas 7 y 8 del código 19 tenemos que se le asigna a **j**, **k** y **m** con 1, 2 y 3 respectivamente.

En la línea 3 se declara un tipo nuevo de variable del tipo **int \***. Esto se denomina una variable apuntador. La **\*** significa que la variable es de tipo apuntador, el nombre asociado, en este caso **int**, muestra a que tipo de información apunta la variable.

Luego se le asigna a **ptr** la dirección de **k** en la línea 10. Esto es seguido por una asignación de **m** a **k**. Lo que sigue es la impresión en la pantalla de los valores que arroja correr este programa. Para poder analizarlo y tratar de entender por completo la teoría de apuntadores, mostramos la salida abajo:

```
[hola@maquina]$ ./apuntador
```

```

j tiene el valor 1 y esta almacenado en 0x8049744
k tiene el valor 3 y esta almacenado en 0x804974c
m tiene el valor 3 y esta almacenado en 0x8049750
ptr tiene el valor 0x804974c y esta almacenado en 0x8049748
El valor del entero apuntado por ptr es 3

```

Como pueden apreciar, los valores que resultan de la corrida del programa son los deseados. Lo importante notar es la diferencia entre la asignación hecha a **ptr** en la línea 10 y la posterior asignación de **k** a **m**. Esto refuerza

la idea de que una variable tipo apuntador sólo tiene la información sobre dónde está el contenido de algo ya que de la primera asignación de 2 a **k**, éste se convierte en 3 y cambia el contenido al que apunta también **ptr**. Esto se puede ver en la línea de salida número 4 donde la dirección en memoria es igual a la dirección de la variable **k**.

En el código 19 también se muestran los usos de **&** y **\*** en la línea 13 hasta la 18.

La idea de apuntadores no es fácil de entender y sólo con la práctica se puede reforzar este concepto. Tómese un tiempo para digerir la información de esta sección antes de pasar más adelante.

## 7.2. Apuntadores y estructuras de datos

Las estructuras de datos son esenciales para poder realizar cálculos y cosas interesantes en los programas de computadora. En el caso del lenguaje de programación C las estructuras de datos están completamente ligadas a la idea y al uso de los apuntadores.

Si bien existe un capítulo más extenso sobre estructuras de datos en esta guía, esta sección se encargará de tratar de unir la idea de apuntadores a la de estructuras de datos. Comenzaremos con la estructura de datos más básica: los arreglos.

### 7.2.1. Arreglos

El arreglo es la estructura de datos más básica en computación. La idea de un arreglo es sencilla. Imagínese los vagones de un tren, cada uno con un número, siendo 0 el primer vagón. En cada vagón se puede meter algún contenido, pero todos los contenidos del tren tienen que ser del mismo *tipo*.

A diferencia de otros lenguajes de programación los arreglos en C deben ser inicializados sabiendo su tamaño máximo. Un ejemplo es:

```
int miarreglo[10];
```

En este ejemplo se crea un arreglo de enteros con 10 casillas, del 0 al 9. Otro ejemplo es:

```
int miarreglo[] = {4, 5, 7, 8};
```

En este ejemplo se crea un arreglo de 4 casillas, del 0 al 3, con contenido de 4, 5, 7 y 8.

Abstrayéndonos al nivel de C, se puede decir que un arreglo de tamaño N, debe reservar N espacios de memoria de un tamaño deseado. Parece lógico que estos espacios de memoria se reservasen de forma continua. En efecto esto es lo que pasa en C.

Para poder ver como entra en juego los apuntadores mostramos el ejemplo de como iterar por las casillas de un arreglo.

```
#include <stdio.h>

int miarreglo [] = {1,23,17,4,-5,100};

int main(void) {

    int i;

    printf("\n\n");
    for (i = 0; i < 6; i++) {
        printf("miarreglo[%d] = %d\n", i, miarreglo[i])
        ;
    }
    return 0;
}
```

Código 28: Iteración de un arreglo mediante casillas

En este ejemplo, se usa la teoria de casillas para iterar por el arreglo. El for es simple y imprime en la pantalla el valor de *miarreglo[x]* donde x toma un valor del 0 al 5 (por lo tanto son 6 valores).

Pero, ¿si la información en memoria está puesta secuencialmente, no habrá una forma de usar apuntadores? En C todo es posible.

Si se recuerdan, en la sección anterior, se habló sobre los tamaños de las variables y sobre su reservación en memoria al momento de declararlas (inicializadas o no). Si se declara un arreglo de enteros (int) *miarreglo* de 6 casillas, esto significa que el compilar está reservando  $6 * \text{sizeof}(\text{int})$  (ver código 2, página 9) espacio en memoria. Con un poco de aritmética simple nos damos cuenta que cada casilla del arreglo toma  $\text{sizeof}(\text{int})$  y que si hacemos  $\&\text{miarreglo}[0]$ , ¿nos daría la dirección en memoria de la primera casilla!

Si nos queremos entusiasmar más podemos pensar que si tenemos la dirección de  $\&\text{miarreglo}[0]$  y sumamos un  $\text{sizeof}(\text{int})$ , ¿nos daría la dirección de la segunda casilla del arreglo! En efecto es así. Si nos ponemos a pensar, esto sería un poco tedioso para hacer, ya que implica una aritmética en hexadecimal (es la forma en la que se clasifican los espacios de memoria) por lo que

no es intuitivo incluso para el programador más experimentado.

Cuando uno crea un tipo apuntador en C, aparte de declarar que la variable es de tipo apuntador con `*` se le asigna un tipo de variable (`int`, `float`, `char`, etc). Esto se hace no solo por la legibilidad del código sino también para poder usar los apuntadores en contextos más variados, como para iterar dentro de un arreglo. Como el apuntador tiene asignado un tipo y todas las variables en C pueden participar en operaciones aritméticas, es posible lo que se muestra abajo:

```
1 #include <stdio.h>
2
3 int miarreglo [] = {1,23,17,4,-5,100};
4 int *ptr;
5
6 int main(void) {
7
8     int i;
9     ptr = &miarreglo [0];
10
11     printf("\n\n");
12     for (i = 0; i < 6; i++) {
13         printf("ptr + %d = %d\n", i, *(ptr + i));
14     }
15     return 0;
16 }
```

Código 29: Iteración de un arreglo mediante apuntador auxiliar

Lo interesante de este código es que tenemos algo equivalente al código previo, pero de una forma completamente nueva. Lo importante de este código es la línea 14. Dentro del `printf` hay una expresión sumamente bonita que dice `*(ptr + i)`. Si nos recordamos de la sección anterior `*ptr` nos llevaba al contenido (valor) a que nos apunta `ptr`. Ahora si le sumo un número entero a `ptr` antes de pedir el contenido, ¿qué es lo que está pasando? La respuesta viene dada por la declaración del apuntador con un tipo.

Si tenemos un apuntador tipo entero (como es el caso arriba) el espacio de ese valor va a venir dado por `sizeof(int)`. Entonces si a `ptr` se le agrega el `sizeof(int)`, pasaría a otro espacio de memoria donde *podría* comenzar otro valor de entero. Aunque esto no es recomendable en muchas ocasiones, en el caso de arreglos es perfecto. Para facilitar este cálculo, C viene incorporado con *aritmética de apuntadores*. Esto significa que dado un apuntador declarado:

```
hola *ptr;
```

La expresión:

```
ptr + 1
```

se resolvería con la ecuación:

```
ptr ahora apuntar a = direccion ptr + sizeof(hola)
```

para una variable “hola” cualquiera. Cabe destacar que las operaciones:

```
ptr++
++ptr
ptr--
--ptr
```

son también válidas dentro de C. Ha que tener cuidado con estas ya que su ejecución cambia en que momento de la ejecución se le agrega (o resta) al apuntador. Recuerdense que si uno por error intenta acceder a un espacio de memoria no reservado, el programa explota, en el caso de acceder a un espacio reservado para otra variable, pueden pasar cosas interesantes, pero difíciles de corregir.

Si han entendido el concepto hasta ahora, se preguntarán cómo es que con un arreglo puedo hacer **miarreglo[x]** donde x es un número definido dentro del arreglo y cómo es que puedo usar un apuntador para hacer exactamente lo mismo. Esto se debe a que la declaración de una variable tipo arreglo es en verdad un apuntador.

Para demostrarlo tenemos este último pedazo de código:

```
#include<stdio.h>

int miarreglo [] = {1,23,17,4,-5,100};

int main(void) {

    int i;

    printf("\n\n");
    for (i = 0; i < 6; i++) {
        printf("miarreglo[%d] = %d\n", i, *(miarreglo +
            i));
    }
}
```

Figura 1: Lista enlazada simple

```
}  
    return 0;  
}
```

Código 30: Iteración en un arreglo usando el arreglo como apuntador

Este código es completamente análogo al código 21 en la página 42.

Nuevamente se sugiere sentarse a pensar sobre esta subsección, tratando de entender en cada ejemplo dado lo que está pasando, paso a paso. Una vez que quede claro este concepto podemos pasar a las estructuras más avanzadas.

### 7.2.2. Lista enlazada

Antes de finalizar la parte de apuntadores sería interesante poder crear una estructura básica utilizando los conocimientos adquiridos de apuntadores y estructuras elementales.

«“<.mine ===== La idea de una lista enlazada es muy simple como lo refleja el siguiente dibujo:

»”Lo que se tiene es una estructura básica tipo registro, donde el nodo contiene una información y la información de donde se encuentra el próximo nodo.

»”Para poder comenzar a crear una lista necesitamos un registro primario que luego pueda convertirse en nuestro nodo. Usando nuestro conocimiento de structs y apuntadores, podemos hacer un primer intento:

»”Esta construcción, como se vió en el capítulo de estructuras declara un registro que luego se puede declarar como **nodo** que va a contener dos tipos de datos. Un espacio para guardar la información de un entero y la información de dónde encontrar el próximo nodo en la memoria.

»”Para poder utilizar nuestro nuevo registro de una forma interesante tenemos que poder insertar información nueva a nuestra lista, eliminar, buscar e imprimir el contenido de la misma. Esto no es difícil, y lo trataremos de desglosar de la mejor manera posible.

»”Para crear un nodo nuevo en un programa, es suficiente hacer:

»”La confusión más común es por qué se crea un apuntador a nodo y no simplemente un nodo. Esto depende fuertemente de como se quiera tratar el nodo dentro del programa. En este caso, se quiere hacer las modificaciones sobre un nodo (y posteriormente de una lista) en forma de función. Debido a esto es preferible usar un apuntador a nodo.

»'Para poder agregar al nodo un valor, como está en NULL, se debe inicializar la variable. Como se quiere reservar un espacio dinámico para nuestra estructura, se requiere de un malloc del tamaño de nodo. Una vez reservado el espacio, se debería asignar un valor al contenedor que tiene el nodo. Todo esto se puede hacer, creando una función:

```
»""»>.r51
```

### 7.3. Apuntadores y strings

En C no existe la noción de String como en otros lenguajes de programación. Esto se debe a la idea de que C está lo más cerca del lenguaje ensamblador sin dificultar la tarea de hacer cosas de alto nivel. Por lo tanto en C no existe la idea de Strings sino de arreglos de caracteres.

En C, el texto se declara como un arreglo de caracteres y por lo general son inicializados de la forma:

```
char mistring[40] = "Hola"
```

Esto crea un arreglo de 40 casillas de tamaño sizeof(char), donde los primeros 5 caracteres son **Hola\0**. El resto del espacio estaría completamente en blanco. El famoso \0 es automáticamente agregado al arreglo si se usan doble comillas en la declaración del arreglo. Este carácter no sería impreso en un printf pero muestra el final del espacio utilizado en un arreglo de caracteres.

```
#include<stdio.h>

int main() {
    char contenedor[50] = "Hola esto es una prueba";
    char contenedor2[50];

    char *pA; /* apuntador a un tipo char */
    char *pB; /* apuntador a otro tipo char */
    pA = contenedor; /* apunta pA a contenedor */
    printf("%s -> Muestra a lo que apunta tiene
           contenedor\n", contenedor);
    printf("%s -> Muestra a lo que apunta pA\n", pA);

    pB = contenedor2; /* apunta pB a contenedor2 */
    printf("\n");

    while(*pA != '\0') {
```

```

        *pB++ = *pA++;
    }

    *pB = '\\0';
    printf("%s -> Muestra a lo que apunta contenedor2
        al final de todo.\\n",
        contenedor2);
    return(0);
}

```

Código 31: Declarando y imprimiendo un arreglo de char (String)

Lo importante es ver aquí cómo la variable %s en printf, imprime el arreglo de char sin problema. También hay un fuerte uso de apuntadores y de sus asignaciones. Fíjese bien lo que pasa dentro del ciclo del while. Aquí se está asignando a cada casilla del contenedor2 la información de la variable contenedor. También muy importante es el agregado del “\0” después de haber copiado carácter por carácter.

Si bien esto no es como se copiaría un arreglo normalmente en C, es una demostración muy interesante de como los arreglos de char, los apuntadores y los strings están unidos en este lenguaje.

## 8. Herramientas

### 8.1. Makefile

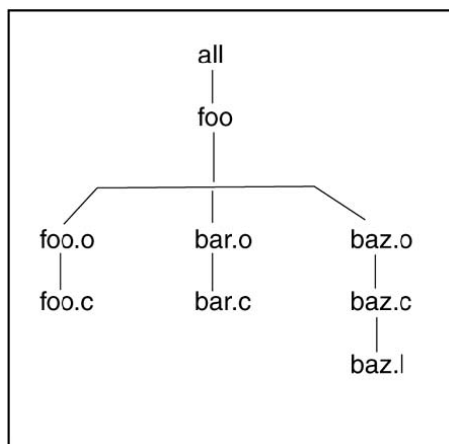
Un archivo **Makefile** es una especificación de dependencias entre archivos y de cómo resolverlas para lograr un objetivo global. Estos archivos son procesados por la utilidad *make*, disponible en cualquier sistema UNIX.

El programa *make* actualiza todos los objetivos a través de la actualización de todas sus dependencias. Un grafo potencialmente complejo es formado cuando se realiza el procesamiento de estos archivos. Un ejemplo sencillo es:

```
all: foo
foo: foo.o bar.o baz.o
.c.o:
    $(CC) $(CFLAGS) -c $< -o $@@
.l.c:
    $(LEX) $< && mv lex.yy.c $@@
```

Código 32: Ejemplo Sencillo Makefile

El grafo de fuentes asociado a este ejemplo es:



Generalmente, todos los objetivos se asumen nombres de archivos, y las reglas deben especificar cómo crear o actualizar estos archivos.

Cuando se llega a los nodos hojas, como se ve en el gráfico previo, el Makefile debe incluir un conjunto de comandos de shell que permite generar o actualizar estos archivos. La actualización de los objetivos significa que el tiempo de última modificación de las dependencias sea más reciente que el objetivo.

Igualmente, se pueden especificar reglas que serán ejecutadas incondicionalmente, por ejemplo:

```
clean:
  rm *.o core
```

Esto indica que al ejecutar el comando `make clean`, se borrarán los archivos con extensión ".o" y el llamado "core".

Los comentarios de un archivo Makefile comienzan con el carácter "#" hasta el final de la línea. El siguiente ejemplo muestra tres objetivos individuales con dependencias individuales:

```
target1: dep1 dep2 ... depN
<tab> cmd1
<tab> cmd2
<tab> ...
<tab> cmdN
```

```
target2: dep4 dep5
<tab> cmd1
<tab> cmd2
```

```
dep4 dep5:
<tab> cmd1
```

Los objetivos deben estar al comienzo de una línea y son seguidos de dos puntos. Luego sigue un espacio en blanco y el conjunto de dependencias separadas por un espacio en blanco. En la línea siguiente van los comandos de *shell* que deben ser ejecutados, es importante conocer que ésta línea debe tener como prefijo un espacio de tabulador, este es un error muy común en nuevos usuarios. Es decir, *no se aceptan varios espacios en blanco, solo un tab*.

Además, en estos archivos se pueden usar ciertas macros, éstas empiezan con un signo de dólar. Por ejemplo: `$(CC) $(CFLAGS) -c $< -o $@`. Aquí la forma sintáctica "\$(..)" indica expansión de variables. Una variable se puede definir de la siguiente forma: `VAR=VALOR`,

```
CC=gcc
```

Al ser procesado, donde se haya especificado la variable `$(CC)` se sustituye literalmente con el valor "gcc". Make tiene un conjunto de variables definidas por defecto, el valor primario de `$(CC)` es "cc".

Las macros más comunes son `$$` y `$(..)`. Éstas representan los nombres de los objetivos y la primera dependencia de la regla en la que aparece. En el siguiente Makefile se puede ilustrar,

```
all: dummy
```

```
@echo "$@" depends on $<"  
dummy:  
    touch $@"
```

Código 33: Ejemplo Sencillo Makefile

Al ejecutar el comando "make", la salida es:

```
$ make  
touch dummy  
all depends on dummy
```

## 8.2. *Debugging* con gdb

Al escribir un programa –en cualquier lenguaje– suele ocurrir que cometemos errores. Los programas escritos pueden tener un comportamiento diferente al que esperábamos debido a que nuestro algoritmo era incorrecto o que nos equivocamos al escribirlo. El proceso de “depuración”, más comúnmente conocido como *debugging* consiste en la búsqueda y eliminación de errores en un programa ya escrito.

Dado que realizar debugging de un programa puede llegar a ser una tarea muy tediosa, existe una variedad de herramientas que ayudan en el proceso. En particular está disponible **GDB**, un debugger que forma parte del proyecto GNU. Éste permite examinar la ejecución de un programa mientras está ejecutándose o al haber terminado su ejecución. Permite detener el programa en puntos especificados para observar su estado e intentar localizar la fuente de cualquier error detectado.

Para que nuestro programa ejecutable contenga información de debugging que nos ayude durante el uso de GDB, debemos compilar usando la opción *-ggdb* de **gcc**. Por ejemplo, podemos hacer:

```
gcc o bugs -ggdb bugs.c
```

Para examinar este programa con gdb hacemos:

```
gdb bugs
```

Al hacer esto estaremos frente a un *prompt* de gdb en el cual podemos escribir comandos. Hay una gran cantidad de comandos disponibles; los usuarios más experimentados tienen muchas posibilidades al utilizar esta herramienta. Sin embargo, conociendo un pequeño conjunto de comandos importantes podemos aprovecharla. Por supuesto, la mejor manera de aprenderlos es usándolos. Experimente con los siguientes comandos. Puede utilizar el comando *help* para obtener más ayuda sobre cada uno. Encuentre los errores en el programa 35, que intenta encontrar el menor número  $n$  tal que  $\text{fib}(n)$  es mayor que 10000000.

- `break`
- `print`
- `display`
- `bt`
- `list`

- run
- set args
- next
- step

```
#include <stdio.h>

int fib(unsigned int n)
{
    if (n == 0)
        return 0;

    if (n = 1)
        return 1;

    return fib(n-1)+fib(n-2);
}

int main(int argc, char** argv)
{
    int i = 0, encontrado = 0;

    while (encontrado = 0)
    {
        if (fib(i) > 10000000)
        {
            encontrado = 1;
        }
    }

    printf("fib(%d) = %d\n", i, fib(i));

    return 0;
}
```

Código 34: programa incorrecto

Para empezar podemos definir un *breakpoint* al entrar a main. Esto hará que el programa se detenga ahí, lo que nos permite ver la ejecución paso a paso desde el principio. Esto se puede hacer escribiendo

`break main`

o, dado que `main` empieza en la línea 14,

`break 14`

Hecho esto, escribimos

`run`

para iniciar la ejecución del programa. Éste se detendrá inmediatamente, permitiéndonos darle más comandos a GDB. Podemos usar los comandos *print* y *display* para ver el contenido de una variable dada. *next* y *step* avanzan un paso en la ejecución.

## 9. Ejercicios

1. Escriba una función *strindex(s,t)* que retorne la posición de la ocurrencia ubicada más a la derecha, del elemento *t* en el string *s*. En caso de que haya ninguna ocurrencia retornar cero.
2. Escriba una función similar a la estudiada previamente *atof* para manejar la notación científica de la forma: 123.45e-6, cuando un punto flotante este seguido de una *e* o *E* y opcionalmente un signo en el exponente.
3. Defina un macro *swap(t,x,y)* que intercambie dos argumentos, *x* e *y*, de tipo *t*.
4. Escriba un programa para comparar dos archivos, imprimiendo las primero línea donde difieren.
5. Modifique el programa de *Grep propio* dado en capítulos previos, que tomen como entrada de parámetros un conjunto de nombres de archivos y busque el patrón en todos esos archivos.
6. Escriba un programa que imprima en pantalla un conjunto de archivos, entre cada archivo debe imprimir una línea en blanco y se debe indicar el nombre del archivo correspondiente en la salida.