Tutorial de Lenguaje C

Departamento de Informática
Universidad Nacional de San Luis
Autor: Dr. Carlos Kavka

Índice

1. Introducción

El lenguaje C es un lenguaje de programación que fue definido por Dennis Ritchie en los laboratorios de la AT&T Bell en 1972. Fue definido en principio para el sistema operativo Unix, pero se ha extendido a casi todos los sistemas hoy en día. De hecho, el diseño de muchos lenguajes más modernos se han basado en las características del lenguaje C.

El lenguaje C es un lenguaje de programación de propósito general, que permite la escritura de programas compactos, eficientes y de alta portabilidad. Provee al programador de un medio conveniente y efectivo para acceder a los recursos de la computadora. Es un lenguaje de suficiente alto nivel para hacer los programas portables y legibles, pero de relativo bajo nivel para adecuarse al hardware particular.

C es un lenguaje muy simple, con un conjunto de operadores muy bien diseñados. No provee, por ejemplo, operadores para trabajar con strings ni arreglos en forma completa, no provee mecanismos de administración dinámica de memoria, ni forma de trabajar con archivos. Es decir, el lenguaje C es muy pequeño y fácil de describir. Todas las operaciones adicionales tales como las mencionadas, cuya falta haría al lenguaje C muy restringido en sus aplicaciones, son provistas a través de funciones presentes en una biblioteca que está incluso escrita en el mismo lenguaje C.

El lenguaje fue documentado completamente en 1977 en el libro "El lenguaje de programación C", de Kernighan y Ritchie. Fue de hecho el primer estándar. A medida que C fue creciendo en popularidad y fue implementado en distintos ambientes, perdió su portabilidad, apareciendo una serie de variantes del lenguaje. En el año 1988 el ANSI (American National Standards Institute) definió un nuevo estándar corrigiendo algunos errores en su diseño original, y agregando algunas características que demostraron ser necesarias. El lenguaje, conocido como ANSI C, es de total aceptación hoy en día.

2. El primer programa

La mejor forma de comenzar a estudiar C es viendo algunos ejemplos. De hecho, todo programador C escribió alguna vez el famoso programa que se muestra a continuación, cuyo único objetivo es imprimir las palabras "Hola mundo" en la pantalla.

```
  #include <stdio.h>

  int main() {

    printf("Hola mundo\n");
    return(0);
}
```

Los programas en C están compuestos por funciones. Normalmente, se tiene la libertad de dar cualquier nombre a las funciones pero la función `main`, cuya definición aparece en este pequeño programa, es una función especial: es el punto donde comienza le ejecución del programa.

El encabezamiento de la definición de la función:

```
  int main()
```

indica, en este ejemplo, que la función `main` no tiene argumentos [1]. Aquí, main está definida para ser una función que no espera argumentos, lo cual se especifica a través de los paréntesis que no encierran ninguna declaración de parámetros formales. El encabezamiento también indica que la función retorna un valor entero (`int` en la terminología de C).

Aquí bien cabe la siguiente pregunta: Si la función `main` es el programa principal, ¿cómo es posible que retorne un valor o tome argumentos?. Los argumentos al programa principal, o función `main` serán tratados en una sección posterior. El valor retornado, un entero en este caso, es un valor que es pasado por el programa

[1] En este tutorial se utilizará indistintamente la palabra **argumento** y **parámetro**

al sistema operativo cuando finaliza la ejecución del programa. Este valor es utilizado por el sistema operativo para determinar si el programa finalizó en forma correcta o se produjo algún error. Por convención, el valor de retorno que indica que el programa finaliza exitosamente es 0. Un valor distinto de 0 indica que el programa no pudo terminar correctamente dado que se produjo algún tipo de error. Este valor es retornado al sistema operativo a través de la sentencia `return`.

```
return 0;
```

La función `printf` es una función de la biblioteca que imprime sus argumentos en la pantalla (mas formalmente se definirá donde imprime en una sección posterior). En este caso, su único argumento es el string `"Hola mundo\n"`, el cual será entonces impreso en la pantalla. La secuencia `\n` que aparece al final del string representa el caracter de nueva línea o *newline*, que especifica que las futuras impresiones deberán comenzar en el margen izquierdo de la siguiente línea.

El mismo efecto podría haberse alcanzado con la siguiente secuencia de invocaciones a `printf`:

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

Al comienzo del programa aparece una línea con el siguiente contenido:

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

Es una instrucción para el preprocesador (se detallará más adelante), en la que se indica que se deben incluir las definiciones de la biblioteca de entrada/salida estándar (standard input output). Esta instrucción es la que habilita que funciones tales como `printf` puedan ser utilizadas. Recuerde que las funciones de entrada/salida no pertenecen al lenguaje, estando su definición en bibliotecas.

Pregunta 1 ¿Cuál es la salida producida por las siguientes sentencias?.

```
printf("Esta es una ");
printf("prueba\n");
```

Pregunta 2 El siguiente programa:

```
int main() {
  printf("otra prueba\n");
  return(0);
}
```

al ser compilado da el error que se muestra a continuación. ¿Por qué?.

```
Error: la funcion printf no esta   definida.
```

La sección 7.1 está destinada a consolidar y ampliar el tema de funciones para en la sección 7.4 ejemplicar con un programa más elaborado que el planteado en esta sección.

3. Tipos de datos

3.1. Nombres de variables y declaraciones

La sentencia de declaración es una de las características más importantes de C. Mediante la sentencia

```
int num;
```

se declaran dos aspectos: primero que en algún sitio de la función se utilizará una "variable" denominada **num**. En segundo lugar, el prefijo **int** identifica a *num* como un entero, es decir un número sin decimales. El compilador utiliza esta información para reservar un espacio de almacenamiento en memoria, adecuado para la variable *num*. El símbolo punto y coma del final de línea identifica ésta como una sentencia o instrucción C. En C es obligatorio declarar todas las variables que se utilizan: se debe suministrar una lista de todas las variables que se usarán más adelante, indicando en cada una de ellas a que "tipo" pertenecen. El lenguaje C maneja varias clases o tipos de datos. El hecho de declarar permite al ordenador almacenar, localizar e interpretar adecuadamente el dato. A continuación se explican los principales tipos de datos.

3.2. El tipo entero

El tipo entero básico de C es el que se ha estado utilizando en los ejemplos. Se requiere que tenga al menos 16 bits (es decir que su rango sea al menos -32768...+32767), y que represente el tamaño natural de la palabra en la computadora. Esto significa que en un procesador cuyos registros aritméticos son de 16 bits, se espera que los enteros tengan 16 bits, en un procesador con registros de 32 bits, se espera que los enteros tengan 32 bits. La razón de esta decisión es la seguridad de que las operaciones con enteros se realizarán con la mayor eficiencia posible.

La siguiente es la declaración de tres variables enteras en C. Notar que la última es inicializada en la misma declaración:

```
int i,j;
int var = 12;
```

Utilizando los calificadores `short` y `long` es posible declarar enteros con menor o mayor rango respectivamente.

```
short int a;
long int b;
```

El rango de la variable entera `a` no será mayor que el de un entero convencional, y el de la variable entera `b` no será menor al de un entero convencional. Cada implementación definirá el tamaño exacto.

También es posible declarar variables enteras que pueden tomar sólo valores positivos, ampliando de esta manera el rango de valores positivos que pueden tomar:

```
unsigned int x;
unsigned short int y;
```

Si el rango de una variable entera es de -32768...+32767, el de la variable `x` será de 0...65535.

No es necesario usar la palabra `int` cuando se realizan estas declaraciones. Las declaraciones anteriores podrían haberse escrito:

```
short a;
long b;
unsigned x;
unsigned short y;
```

Pregunta 3 ¿Es posible que en una implementación particular, un `short`, un `long` y un `int` tengan todos el mismo tamaño?

3.3. El tipo caracter

El tipo caracter `char` permite representar caracteres individuales. Por ejemplo, se declara una variable de tipo `char`:

```
char v;
```

se le pueden asignar caracteres:

```
v = 'A';
```

Sin embargo, en C no existe una diferencia real entre los caracteres y los enteros. De hecho, el tipo `char` es un caso particular de un entero de un byte. La asignación anterior podría perfectamente haberse escrito:

```
v = 65;
```

dado que el código ASCII del caracter `A` es 65.

Se puede notar la total equivalencia si se imprime el valor de la variable `v` como entero y como caracter. El especificador de formato para caracteres es `%c`.

```
printf("el codigo ASCII de %c es %d\n",v,v);
```

Se obtendrá impreso:

```
el codigo ASCII de A es 65
```

Notar que el valor de la misma variable es impreso la primera vez como caracter y la segunda como entero.

Dado que `v` es un entero (aunque bastante pequeño), se puede operar libremente con él:

```
int i;
char v = 'A';

i = v * 2;
```

En el ejemplo se asigna a la variable `i` el valor 130 (65 * 2).

Pregunta 4 ¿Cuál es la salida del siguiente trozo de programa?

```
char v = 'A';

printf("%c %c %c",v,v+1,v+2);
```

La función `getchar` de la biblioteca estándar de entrada/salida retorna el caracter que ha sido ingresado por teclado. Es común utilizar una descripción denominada prototipo cada vez que se explica el uso de una función. En este caso, el prototipo de la función `getchar` es el siguiente:

```
int getchar();
```

El prototipo está indicando que la función no toma argumentos y retorna un entero. Parecería que existe una contradicción dado que hemos dicho que retorna un caracter ingresado por el teclado. La contradicción no es tal, dado que existe una relación directa entre los caracteres y los enteros. De hecho, la función retorna el código ASCII del caracter ingresado por teclado, y también algunos otros valores enteros (fuera del rango de los caracteres ASCII) para indicar la ocurrencia de otros eventos, tales como error en el dispositivo de entrada, fin de archivo, etc. El valor fin de archivo es utilizado para indicar el fin de la entrada y puede ser generado en los sistemas Unix con control-D y en los sistemas MS-DOS con control-Z. La constante `EOF` representa este valor.

Si se presionan los caracteres A,a y Z, el siguiente trozo de programa:

```
int v;

v = getchar();
printf("%c %d ",v,v);
v = getchar();
printf("%c %d ",v,v);
v = getchar();
printf("%c %d ",v,v);
```

imprimirá:

```
A 65 a 97 Z 90
```

3.4. Constantes enteras

Las constantes enteras se pueden escribir en distintas notaciones. Una secuencia de dígitos que no comienza con 0 es considerada un entero decimal, si comienza con un 0 (cero) es considerada una constante octal, y si comienza con la secuencia `0x` (cero equis) una constante hexadecimal. Por ejemplo:

```
int i;

i = 255;                /* 255 decimal */
i = 0xff;               /* ff hexadecimal = 255 decimal */
i = 0xf3;               /* f3 hexadecimal = 243 decimal */
i = 027;                /* 27 octal = 23 decimal */
```

También es posible definir constantes `long`, las que deben ser utilizadas cuando el valor que se desea especificar está fuera del rango soportado por el entero común. Se escribe con una letra l (ele) o L siguiendo el número. Por ejemplo, suponiendo que en la implementación particular un `int` tiene 16 bits y un `long` 32 bits:

```
long x;

x = 1000;               /* en el rango entero */
x = 123456789L;         /* fuera del rango entero */
x = 1000L;              /* la L no es necesaria, pero tampoco es un error */
```

También existen constantes `unsigned`, las que se especifican con una u o U siguiendo al número. Por ejemplo:

```
unsigned y;

y = 455u;
```

Pregunta 5 ¿Cuál es el valor asignado a las variables enteras?

```
int a,b;

a = 0x28 + 010 + 10;
b = 0xff + 'A';
```

Pregunta 6 Identifique el error en el siguiente trozo de programa:

```
int a;

a = 08 + 02;
```

Pregunta 7 El siguiente trozo de programa contiene dos asignaciones para la variable `x`. ¿Cuáles de ellas son válidas si los enteros se representan con 16 bits?.

```
unsigned x;

x = 50000;
x = 50000u;
```

3.5. Combinando enteros

Al existir varias clases de enteros, deben existir reglas que permitan determinar cual es el tipo y el valor del resultado de operaciones que los combinen. Ellas son:

1. Los caracteres (`char`) y los enteros cortos (`short`) son convertidos a enteros comunes (`int`) antes de evaluar.
2. Si en la expresión aparece un entero largo (`long`) el otro argumento es convertido a entero largo (`long`).
3. Si aparece un entero sin signo (`unsigned`) el otro argumento es convertido a entero sin signo (`unsigned`).

Suponga que se tienen las siguientes definiciones:

```
int a = 5;
long b = 50000L;
char c = 'A';
unsigned d = 33;              /* o 33u, es lo mismo */
```

Las siguientes expresiones se evalúan de la siguiente forma:

expresión	resultado	tipo	reglas usadas
`a + 5`	10	`int`	
`a + b`	50005	`long`	2
`a + c`	70	`int`	1
`a + b + c`	50070	`long`	1 2
`a + d`	38	`unsigned`	3

Se debe tener especial cuidado cuando se trabaja con enteros sin signo (`unsigned`), ya que a veces se pueden obtener resultados no esperados. Por ejemplo, el resultado de la evaluación de la expresión `2u - 3` es 65535 en una computadora de 16 bits. La razón de este hecho radica en que al ser uno de los argumentos `unsigned`, por aplicación de la regla 3, el otro es convertido a `unsigned` y el resultado debe ser `unsigned` también. Observar como las operaciones se realizan en notación binaria:

```
      2u   0000000000000010
-     3u   0000000000000011
  65535u   1111111111111111
```

El último valor binario con signo (en notación complemento a dos) es interpretado como -1, pero por aplicación de las reglas debe ser considerado como un `unsigned`, por lo que el resultado es 65535.

Pregunta 8 ¿Cuál es el tipo y el resultado de la evaluación de las siguientes expresiones en una computadora de 16 bits?

```
10 + 'A' + 5u
50000u + 1
50000 + 1
```

3.6. El tipo flotante

Existen tres tipos de flotantes (o reales) en C. El tipo flotante estándar se denomina `float`, y las versiones de mayor precisión `double` y `long double`. Al igual que con los enteros, la precisión de cada uno depende de la implementación. Sólo se asegura que un `double` no tiene menor precisión que un `float`, y que un `long double` no tiene menor precisión que un `double`.

Un ejemplo de uso de flotantes es el siguiente [1]:

```
float f,g = 5.2;


f = g + 1.1;
printf("%f\n",f);     /* imprime 6.3 */
```

3.7. Combinando flotantes con enteros

Se debe tener cuidado cuando se mezclan enteros con flotantes dado que se pueden obtener resultados inesperados en algunos casos. Considere el siguiente ejemplo, el cual imprime 3 y no 3.5 como podría esperarse

```
float f;

f = 7 / 2;
printf("%f\n",f);    /* imprime 3 */
```

La razón es que ambos argumentos de la división son enteros. Al ser ambos enteros, la operación de *división entera* es evaluada en dos argumentos enteros y el resultado es entonces entero. Una posible solución para este problema es forzar a que uno de los argumentos sea flotante. C interpretará entonces que se desea realizar una división de flotantes, por ejemplo:

```
float f;

f = 7 / 2.0;
printf("%f\n",f);    /* imprime 3.5 */
```

3.8. Conversión de tipos

Tal como se ha visto en la sección 3.5 y en la 3.7, se debe tener especial cuidado cuando se combinan operadores de tipos distintos. Como regla general, si un argumento "más chico" debe ser operado con uno "más grande", el "más chico" es primero transformado y el resultado corresponde al tipo "más grande". Por ejemplo, si un `int` es sumado a un `long`, el resultado será `long`, dado que es el tipo "más grande".

C permite que valores "más chicos" sean asignados a variables "más grandes" sin objeción, por ejemplo:

```
float f = 3;       /* 3 es un entero y f float */
double g = f;      /* f es float y g double */
```

También se permite que valores de un tipo "más grande" sean asignados a variables más "chicas", aunque dado que esta situación podría dar lugar a errores (ver pregunta al final de la sección), el compilador emite un mensaje de advertencia (warning). Realmente no se recomienda realizar este tipo de operación.

```
long x = 10;
int i = x;        /* permitido pero peligroso */
```

[1] El especificador %f se utiliza para imprimir flotantes

A fin de que el programador tenga una herramienta para indicar la clase de conversión de tipo que desea, el lenguaje provee una construcción que permite forzar un cambio de tipo. Se denomina *cast*, y consiste en especificar el tipo deseado entre paréntesis frente a la expresión que se desea cambiar de tipo. Por ejemplo, para que el compilador no emita una advertencia en el ejemplo anterior:

```
long x = 10;
int i = (int)x;
```

El cast `(int)` fuerza el valor de `x` a tipo entero. Si bien aún podría producirse una pérdida de información, el programador demuestra que es consciente de ello.

Para el caso de la división planteado en la sección 3.7, el problema podría ser resuelto con un cast como sigue:

```
float f;

f = (float)7 / 2;
printf("%f\n",f);    /* imprime 3.5 */
```

Al ser el 7 convertido a `float` por el cast, la operación de división se realiza entre flotantes, y el resultado es 3.5

Pregunta 9 ¿En qué situaciones de asignación de valores de tipo "más grande" en variables de tipo "más chico" se podrían presentar problemas? Dé un ejemplo.

Pregunta 10 En el último ejemplo se utiliza el cast para forzar el 7 a flotante, aunque ello se hubiera logrado más fácilmente escribiendo 7.0. ¿Sería posible lograr la división flotante en el siguiente caso sin usar un cast?:

```
float f;
int i = 7,j = 2;

f = (float)i / j;
printf("%f\n",f);    /* imprime 3.5 */
```

Pregunta 11 ¿Cuáles son los valores asignados?.

```
int a = 3,b;
float f = 2.0, g;

b = a + (int)f;
g = a / f;
```

4. Operadores

C es un lenguaje que provee un conjunto extremadamente rico de operadores. Es una característica que lo distingue de la mayoría de los lenguajes. Una desventaja, sin embargo, es que contribuyen a aumentar la complejidad.

4.1. Operadores aritméticos

Los operadores aritméticos son los siguientes:

operador	descripción
`+`	suma
`-`	resta
`*`	producto
`/`	división
`%`	módulo

Se debe tener en cuenta que las operaciones entre enteros producen enteros, tal como fue mostrado en los ejemplos de las secciones 3.5 y 3.7.

Cuando un argumento es negativo, la implementación del lenguaje es libre de redondear hacia arriba o hacia abajo. Esto significa que con distintos compiladores es posible obtener resultados distintos para la misma expresión [2]. Por ejemplo, la siguiente expresión puede asignar a `i` el valor -2 o el valor -3:

```
i = -5 / 2;
```

Algo similar ocurre con la operación de módulo, que puede asignar a `i` el valor 1 o -1, quedando el signo del resultado dependiente de la implementación particular.

```
i = -5 % 2;
```

4.2. Operadores relacionales

Los operadores relacionales son los siguientes:

operador	descripción
`<`	menor
`>`	mayor
`<=`	menor o igual
`>=`	mayor o igual
`==`	igual
`!=`	distinto

Una característica distintiva de C es que no existe tipo boolean. Las expresiones relacionales tienen valor entero: el valor de una expresión falsa es 0 y el de una verdadera es 1. Por ejemplo, las siguientes expresiones tienen los siguientes valores

`1 < 5` $\longrightarrow$ 1 (verdadera)
`4 != 4` $\longrightarrow$ 0 (falsa)
`2 == 2` $\longrightarrow$ 1 (verdadera)

Al no existir un tipo boolean, las sentencias del lenguaje que requiren condiciones deben aceptar los enteros producidos por su evaluación. Por regla general, un valor entero distinto de cero es considerado como verdadero, y un valor igual a cero como falso. La siguiente condición:

```
if (i != 0) ...
```

es entonces equivalente a:

```
if (i) ...
```

ya que ambas condiciones son consideradas verdaderas cuando `i` es distinto de cero, y serán ambas falsas sólo cuando `i` sea igual a cero.

Dado que las condiciones retornan valores enteros, pueden ser utilizadas dentro de otras expresiones. Por ejemplo:

[2] Esto constituye un ejemplo no portable, y por lo tanto debería evitarse. Llama la atención que cuando se definió el estándar ANSI no se haya dado una definición a este problema.

```
10 + (3 < 4)     ⟶ 11
40 - (4 == 5)    ⟶ 40
```

Pregunta 12 ¿Cuál es el valor de las siguientes expresiones?.

```
int i = 2,j = 3;

i == (j - 1)
i < (j > 0) + 5
i + (j > 0) + 5
```

4.3. Operadores lógicos

Los operadores lógicos son los siguientes:

operador	descripción
`&&`	and (binario)
`||`	or (binario)
`!`	not (unario)

Estos operadores tienen una característica muy interesante, para los operadores binarios se garantiza que el segundo operando sólo es evaluado si es necesario. Es decir, el segundo operando de un and es evaluado sólo si el primero es verdadero, ya que si fuera falso el resultado de todo el and no depende de la evaluación del segundo. Algo similar ocurre con el or, tal como se muestra a continuación:

```
0 && ...   ⟶ 0
1 || ...   ⟶ 1
```

Por ejemplo,

```
if(i != 0 && j/i > 5) ...
```

es una condición bien definida. Si el valor de `i` es cero, no se evalúa la segunda condición, que podría ser problemática.

Pregunta 13 Indique el resultado de la evaluación de las siguientes expresiones y si son evaluadas en forma completa:

```
int i = 0,j = 3;

i < j && j > 2
i < j && j
i != j || j == 8
i + ((j - 3) || i)
```

4.4. Operadores a nivel de bits

C provee operadores que permiten trabajar directamente con los bits de objetos de datos de tipo entero (en todas sus formas). Son los siguientes:

operador	descripción
`&`	and a nivel de bits
`|`	or a nivel de bits
`^`	or exclusivo
`<<`	shift a izquierda
`>>`	shift a derecha
`~`	complemento

Por ejemplo, la expresión `23 & 26` tiene valor 18. Para que quede más claro, a continuación se muestra como se realiza la operación con los equivalentes binarios:

	23	0 ... 0 1 0 1 1 1
	26	0 ... 0 1 1 0 1 0
&	18	0 ... 0 1 0 0 1 0

En general, es más fácil de visualizar los cálculos cuando se utiliza notación hexadecimal. El ejemplo siguiente utiliza la operación de shift a izquierda:

```
int c = 0x0a;

c = c << 4;
```

El valor que se asigna a `c` es el que tenía anteriormente, desplazado a izquierda cuatro bits, por lo que el nuevo valor será 0xa0. Se muestra el equivalente binario a continuación:

	0x0a	0 ... 0 0 0 0 1 0 1 0
<< 4	0xa0	0 ... 1 0 1 0 0 0 0 0

Las operaciones de complemento, or y or exclusivo se definen en forma similar de acuerdo a sus correspondientes tablas de verdad.

Pregunta 14 Indique la salida producida por el siguiente trozo de programa:

```
char c1 = 0x45,c2 = 0x71;

printf("%x | %x = %x\n",c1,c2,c1 | c2);
```

Pregunta 15 El operador de complemento provee una forma portable de colocar en 1 todos los bits de una variable. La definición:

```
unsigned a = 0xffff;
```

no asegura que todos los bits de la variable `a` queden en 1. Si lo asegura la siguiente definición:

```
unsigned a = ~0;
```

¿Por qué?.

4.5. Operadores de asignación

En C la asignación es una operación y no una sentencia. Si bien se lo ha usado en forma equivalente hasta ahora, el operador de asignación retorna el valor que es asignado. Por ejemplo:

```
int a = 2, b, c;

c = (b = a + 1) + 4;
```

A partir del ejemplo se asigna a la variable `b` el valor 3. Este valor es retornado por la asignación interna, el cual sumado con 4 es asignado finalmente a `c`.

Al ser combinado con otros operadores, da lugar a toda una familia de operadores de asignación, que permite escribir las operaciones en forma mucho más compacta, por ejemplo:

normal	versión compacta
`a = a + b`	`a += b`
`a = a - b`	`a -= b`
`a = a * b`	`a *= b`
`a = a / b`	`a /= b`
`a = a >> 2`	`a >>= 2`
`a = a & 0x21`	`a &= 0x21`

Se puede notar que la economía es evidente cuando los nombres de las variables son más descriptivos, por ejemplo:

```
balance_de_cuenta = balance_de_cuenta - 100;
```

se puede escribir como:

```
balance_de_cuenta -= 100;
```

Pregunta 16 Escriba en versión extendida normal las siguientes expresiones compactas:

```
a |= 0x20;
a <<= (b = 2);
```

4.6. Operadores de incremento y decremento

Para incrementar y decrementar el valor de las variables se pueden utilizar respectivamente los operadores de incremento (`++`) y de decremento (`--`). Por ejemplo las siguientes expresiones son equivalentes:

normal	versión compacta
`i = i + 1`	`i++`
`i = i - 1`	`i--`

Se pueden utilizar en forma prefijo o postfijo, con significados distintos. En forma prefijo primero se realiza la operación de incremento o decremento y luego se entrega el resultado para las otras operaciones. En forma postfijo primero se entrega el resultado de la variable, y luego se incrementa o decrementa el valor de la variable. Por ejemplo, las siguientes expresiones son equivalentes:

versión compacta	normal
`a = i++;`	`a = i;`
	`i = i + 1;`
`a = ++i;`	`i = i + 1;`
	`a = i;`
`a = --i + 5;`	`i = i - 1;`
	`a = i + 5;`
`a = i++ + 5;`	`a = i + 5;`
	`i = i + 1;`

Pregunta 17 Suponga que el valor de la variable `i` es 8 y el de la variable `j` es 5. ¿ Qué valores se asignan a las variables `a`, `b`, `c` y `d`?. Indique la forma extendida de las expresiones.

```
a = i++ - j++;
b = ++i - ++j;
c = (d = i--) + j--;
```

4.7. Operador condicional

C provee un operador muy útil denominado operador condicional (`?:`), que permite expresar operaciones similares a la selección, pero dentro de expresiones. Toma tres expresiones como argumentos. La primera es una condición, la cual una vez evaluada determinará cual de las otras expresiones será evaluada para calcular el resultado. Si la condición es verdadera, la segunda es evaluada, en caso contrario la tercera es evaluada.

Por ejemplo,

```
a = (i < j) ? i : j;
```

asigna a la variable `a` el valor de la variable `i` si su valor es menor que el valor de `j`, y el de `j` en caso contrario. Es decir, en otras palabras, asigna el menor valor.

También se puede utilizar en el primera parte de la asignación:

```
(i != j) ? a : b = 0;
```

asigna el valor 0 a la variable `a` si el valor de la variable `i` es distinto al valor de la variable `j`. Si ambas variables tiene el mismo valor, entonces el valor 0 es asignado a la variable `b`.

Pregunta 18 ¿ Qué valores se asignan a las variables `a`, `b`, `c` y `d`?.

```
i = 2;
j = 3;
a = (i == j) ? i++ : j++;
b = (i != 0) ? j / i : 0;
(i < j) ? c : d = j--;
```

5. Control de secuencia

5.1. La sentencia de selección `if`

La forma general de la sentencia básica de selección `if` es la siguiente:

```
if (<condicion>)
  <sentencia>
```

Si la condición es verdadera (distinta de cero) la sentencia es ejecutada. Por ejemplo:

```
if (ganancia < 0)
  printf("no se gana nada\n");
```

En lugar de una sentencia, se puede poner un bloque formado por sentencias encerradas entre llaves. Por ejemplo:

```
if (ganancia < 0) {
  veces++;
  printf("no se gana nada\n");
}
```

También se puede incorporar una sentencia a ser ejecutada en caso que la condición sea falsa, con la siguiente forma general:

```
if (<condicion>)
  <sentencia1>
else
  <sentencia2>
```

Por ejemplo:

```
if (ganancia < 0)
  printf("no se gana nada\n");
else
  printf("se gana %d pesos\n",ganancia);
```

Pregunta 19 Considere el siguiente fragmento de programa:

```
if (i < 5)
if (j < 8)
  printf("uno\n");
else
  printf("dos\n");
```

Hay dos sentencias `if` y una `else`. ¿A qué `if` corresponde el `else`?.

Pregunta 20 El siguiente programa parece empecinado en que el balance sea cero. Explique la razón:

```
#include <stdio.h>
int main() {
  int balance;

  balance = 1000;
  if (balance = 0)
    printf("el balance es cero\n");
  else
    printf("el balance es %d\n",balance);
  return(0);
}
```

El programa imprime:

```
el balance es 0
```

5.2. La sentencia de selección `switch`

El `switch` es una sentencia de selección con múltiples ramas. Su forma general es la siguiente:

```
switch(<expresion>) {
  case <cte1> : <sentencias>
  case <cte2> : <sentencias>
  ...
  default : <sentencias>
}
```

Primero la expresión es evaluada, y luego el grupo de sentencias asociado a la constante que corresponde al valor de la expresión es ejecutado. Si el valor no coincide con ninguna constante, el grupo de sentencias asociadas a la cláusula `default` es ejecutado.

La cláusula `default` es opcional, por lo que si no se ha especificado, y el valor de la expresión no coincide con ninguna constante, la sentencia `switch` no hace nada.

Es importante aclarar que una vez que una alternativa es seleccionada, las siguientes se continúan ejecutando independientemente del valor de su constante.

Por ejemplo, el siguiente trozo de programa imprime `uno`, `dos` y `tres` si el valor de la variable `i` es 1, imprime `dos` y `tres` si el valor es 2, e imprime `tres` si el valor de la variable `i` es 3.

```
switch (i) {
  case 1: printf("uno\n");
  case 2: printf("dos'\n");
  case 3: printf("tres\n");
}
```

Para evitar este efecto se puede utilizar la sentencia `break`, que fuerza el control a salir de la sentencia `switch`, por lo que las siguientes alternativas no serán ejecutadas. Por ejemplo, el siguiente trozo de programa imprime `uno` si el valor de la variable `i` es 1, imprime `dos` si el valor es 2, e imprime `tres` si el valor de la variable `i` es 3.

```
switch (i) {
  case 1: printf("uno\n");
          break;
  case 2: printf("dos'\n");
          break;
  case 3: printf("tres\n");
}
```

Pregunta 21 ¿Qué valor imprime el siguiente trozo de programa en el caso en que `i` valga 1 inicialmente y en el caso en que valga 3?

```
switch(i) {
  case 1: i += 2;
  case 2: i *= 3;
          break;
  case 3: i -= 1;
  default: i *= 2;
}
printf("%d\n",i);
```

No se permiten múltiples valores para una alternativa, pero su efecto se puede simular aprovechando el hecho que la ejecución continúa cuando se ha encontrado la constante adecuada y no existen sentencias `break`. Por ejemplo:

```
switch (ch) {
  case ',':
  case '.':
  case ';': printf("signo de puntuacion\n");
            break;
  default : printf("no es signo de puntuacion\n");
}
```

Las expresiones deben ser de tipo entero, caracter o enumeración (explicada más adelante).

5.3. La sentencia de iteración `while`

Su forma básica es la siguiente:

```
while (<condicion>)
  <sentencia>
```

La sentencia se ejecutará mientras la condición sea verdadera. Si la condición es falsa inicialmente, no se ejecuta la sentencia y la ejecución continúa luego del fin del `while`.

El siguiente programa cuenta el número de espacios en la entrada, hasta que se genera el caracter de fin de archivo (EOF) por el teclado (ver sección 3.3).

```
/* cuenta el numero de espacios */
#include <stdio.h>

int main() {
  int c,nro = 0;

  c = getchar();                    /* lee primer caracter */
  while (c != EOF) {                /* hasta fin de archivo */
    if (c == ' ') nro++;            /* es un espacio */
    c = getchar();                  /* lee otro caracter */
  }
  printf("numero de espacios = %d\n",nro);
  return(0);
}
```

El programa lee caracteres del teclado y los almacena en la variable `c`. Mientras no se haya leído un caracter `EOF`, incrementa el contador `nro` si el caracter es un espacio. Luego un nuevo caracter es leído y el proceso se repite.

5.4. La sentencia de iteración `do while`

Su forma básica es la siguiente:

```
do
  <sentencia>
while (<condicion>)
```

La sentencia se ejecuta hasta que la condición sea falsa. Notar que se ejecuta al menos una vez, a diferencia del `while`. El mismo ejemplo anterior reescrito con la sentencia `do while` es:

```
/* cuenta el numero de espacios */
#include <stdio.h>

int main() {
  int c,nro = 0;

  do {
    c = getchar();                  /* lee un caracter */
    if (c == ' ') nro++;            /* es un espacio */
  } while (c != EOF);

  printf("numero de espacios = %d\n",nro);
  return(0);
}
```

5.5. La sentencia de iteración `for`

La sentencia `for` es una sentencia muy poderosa que permite reescribir en forma más compacta expresiones comunes del tipo:

```
<sentencia inicial>
while (<condicion>) {
  <sentencia cuerpo>
  <sentencia iteracion>
}
```

como:

```
for(<sentencia inicial>;<condicion>;<sentencia iteracion>)
  <sentencia cuerpo>
```

en las que la sentencia inicial prepara el comienzo de la iteración (inicializa variables, etc), la condición controla si se debe o no realizar una nueva iteración, y la sentencia de iteración es la que realiza las acciones necesarias para entrar en una nueva iteración. Por ejemplo, la siguiente iteración `while` imprime los enteros de 1 a 10:

```
i = 1;
while (i <= 10) {
  printf("%d\n",i);
  i++;
}
```

Se puede reescribir utilizando `for` de la siguiente manera:

```
for(i = 1;i <= 10;i++)
  printf("%d\n",i);
```

La primera expresión del `for` es ejecutada primero, luego la segunda expresión (o condición) es evaluada. Si es verdadera, se ejecuta el cuerpo y luego la tercera expresión. El proceso continúa hasta que la segunda expresión se vuelva falsa.

Las expresiones del `for` pueden ser omitidas, por ejemplo, todos los siguientes trozos de programa son equivalentes al ejemplo anterior:

```
i = 1;
for(;i <= 10;i++)
  printf("%d\n",i);
```

```
for(i = 1;i <= 10;)
  printf("%d\n",i++);
```

```
i = 1;
for(;i <= 10;)
  printf("%d\n",i++);
```

Si se omite la condición se asume verdadera siempre.

La siguiente función calcula el factorial de un entero pasado como argumento. Notar que la función toma un entero y sin embargo devuelve un `long`. Fue definida de esta manera dado que es esperable que se requiera mayor precisión para representar el factorial.

```
long factorial(int n) {
  int i = 2;
  long fact = 1;

  for(;i <= n;i++)
    fact *= i;
  return fact;
}
```

El operador coma (`,`) es muy útil cuando se combina con la sentencia `for`. El operador coma toma dos expresiones como argumento, evalúa las dos, y sólo retorna el valor de la segunda. Es útil para colocar dos expresiones, donde sólo se admite una. Por ejemplo:

```
for(i = 0,j = 5;i < j;i++,j--)
  printf("%d %d\n",i,j);
```

imprimirá:

```
0 5
1 4
2 3
```

Notar que la primera y la última expresión del `for` son reemplazada por dos expresiones cada una.

Pregunta 22 Ejecute la función `factorial` para `n` igual a 0,1,2 y 4.

Pregunta 23 Reescriba el ejemplo dado en la sección 5.3 usando for.

Pregunta 24 El siguiente programa debería imprimir el código ASCII de todas las letras mayúsculas, sin embargo, imprime sólo 91, que no corresponde a ninguna letra. ¿Cuál es la razón?

```
#include<stdio.h>

int main() {
  int c;

  for(c = 'A';c <= 'Z';c++);
    printf("%d\n",c);
  return(0);
}
```

5.6. Las sentencias `break` y `continue`

La sentencia `break` tiene dos usos. Uno ya fue explicado en la sección 5.2 en relación con el `switch`. Cuando se utiliza dentro una sentencia de iteración (`for`, `do while` o `while`), causa que se salga de la iteración.

Por ejemplo, el siguiente programa lee 50 caracteres de la entrada, e imprime el número de vocales.

```
/* cuenta el numero de vocales */
#include <stdio.h>

int main() {
  int i,c,nro = 0;

  for(i = 0;i < 50;i++) {
    c = getchar();                  /* lee un caracter */
    if (c=='a' || c=='e' || c=='i' || c=='o' || c=='u')
      nro++;
  }
  printf("numero de vocales = %d\n",nro);
  return(0);
}
```

Para permitir que el programa termine correctamente si se ingresa el caracter fin de archivo antes de los 50 caracteres, se puede agregar un `break`:

```
/* cuenta el numero de vocales */
#include <stdio.h>
#include <stlib.h>

int main() {
  int i,c,nro = 0;

  for(i = 0;i < 50;i++) {
    c = getchar();                  /* lee un caracter */
    if (c == EOF) break;            /* sale de la iteracion */
    if (c=='a' || c=='e' || c=='i' || c=='o' || c=='u')
      nro++;
  }
  printf("numero de vocales = %d\n",nro);
  return(0);
}
```

Si el caracter EOF es leído, sale de la iteración sin importar si se cumple o no la condición. El número de vocales contadas hasta el momento será impreso.

La sentencia `continue`, que puede ser utilizada únicamente dentro de una iteración, causa que se abandone la iteración corriente, y se comienze una nueva. Podría ser utilizada para ignorar los espacios en el ejemplo, y no realizar entonces su procesamiento. Por ejemplo:

```
/* cuenta el numero de vocales */
#include <stdio.h>
#include <stlib.h>

int main() {
  int i,c,nro = 0;

  for(i = 0;i < 50;i++) {
    c = getchar();                  /* lee un caracter */
    if (c == EOF) break;            /* sale de la iteracion */
    if (c == ' ') continue;         /* vuelve otra vez */
    if (c=='a' || c=='e' || c=='i' || c=='o' || c=='u')
      nro++;
  }
  printf("numero de vocales = %d\n",nro);
  return(0);
}
```

Si el caracter leído es un espacio, vuelve a comenzar otra iteración sin ejecutar la parte restante del cuerpo. Notar que la sentencia `i++` es ejecutada dado que siempre se ejecuta al terminar una vuelta de la iteración, ya sea en forma normal, o porque la terminación ha sido forzada por el `continue`.

Una buena utilización de las sentencias `break` y `continue` evita el uso de ramas enormes en selecciones y el uso de flags en las iteraciones.

Pregunta 25 Determine cuál es el objetivo del siguiente programa:

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

int main() {
  int c;
```

```
    int min = 0,may = 0,chars = 0;

    while (1) {
      c = getchar();

      if (c == EOF) break;
      if (c == ' ' || c == '\n') continue;

      if (c >= 'A' && c <= 'Z') may++;
      if (c >= 'a' && c <= 'z') min++;
      chars++;
    }
    printf("%d %d %d\n",chars,may,min);
    return(0);
  }
```

Pregunta 26 Reesciba el programa anterior sin usar `break` ni `continue`.

6. Objetos de datos estructurados

6.1. Arreglos

Un arreglo `a` de tres enteros se define en C como sigue:

```
int a[3];
```

Esta definición reserva lugar para tres variables enteras que pueden ser accedidas a través de un índice entero:

```
a[0] a[1] a[2]
```

Los subíndices comienzan siempre desde cero, por lo que el índice del último elemento, será el tamaño del arreglo - 1. Por ejemplo, el siguiente programa inicializa todos los elementos del arreglo `a` con cero:

```
int main() {
  int a[3],i;

  for(i = 0;i < 3;i++)
    a[i] = 0;
  return(0);
}
```

Un arreglo puede ser inicializado en la misma definición, por ejemplo:

```
int a[3] = { 4,5,2 };
float b[] = { 1.5,2.1 };
char c[10] = { 'a','b' };
```

El arreglo `a` es un arreglo de tres componentes enteras inicializadas con los valores 4, 5 y 2 respectivamente. El arreglo `b` es un arreglo de dos componentes flotantes inicializadas con los valores 1.5 y 2.1 respectivamente. El tamaño del arreglo `b`, que no fue especificado en la definición se obtiene de los datos. El arreglo `c` es un arreglo de 10 componentes de tipo caracter, de las cuales sólo la primera y la segunda han sido inicializadas, con los valores `a` y `b` respectivamente.

6.2. Estructuras

Las estructuras son conjuntos de una o más variables que pueden ser de igual o distinto tipo, agrupadas bajo un solo nombre. Las estructuras ayudan a organizar datos complicados, en particular dentro de programas grandes, debido a que permiten que a un grupo de variables relacionadas se les trate como una unidad en lugar de como entidades separadas.

Por ejemplo, si se deseara mantener en un programa los datos de un empleado, primero se tendría que identificar los atributos de interés, como ser nombre, domicilio, sueldo, etc.

A diferencia del arreglo, en el que todos los elementos son del mismo tipo, y se acceden a través de un índice, en la estructura los elementos pueden ser de tipos distintos y se acceden a través de un nombre.

La forma general de la declaración de una estructura es la siguiente:

```
struct <nombre-estructura> {
  <tipo> <nombre-campo>
  <tipo> <nombre-campo>
  ...
} <nombre-variable>;
```

Entonces, para el ejemplo antes mencionado, se podría definir la siguiente estructura.

```
struct empleado {
  int codigo;
  float sueldo;
} emp;
```

La palabra clave `struct` declara una estructura. El identificador `empleado` es el nombre de la estructura y puede ser utilizado en declaraciones posteriores. Cada elemento de la estructura se conoce como "miembro" o "campo". Al final de la llave pueden ir identificadores que serán variables del tipo indicado (`empleado`), esto reserva memoria para cada una de las variables. Por ejemplo:

```
struct empleado otro_emp;
```

Para hacer referencia a los campos o miembros de una variable de tipo estructura, se hace de la siguiente manera:

```
<nombre_estructura>.<nombre_campo>
```

.

Por ejemplo:

```
emp.codigo = 8652;
emp.sueldo = 1210.25;
```

El nombre de la estructura se podría haber omitido. Por ejemplo:

```
struct {
  int codigo;
  float sueldo;
} emp;
```

Esta declaración define la variable `emp` de tipo estructura con el formato especificado, pero al no tener nombre la estructura, es imposible definir nuevas variables de este mismo tipo.

Se podría haber omitido el nombre de la variable, en cuyo caso sólo se estaría definiendo el tipo estructura, que podría ser utilizado después para crear las variables. Por ejemplo:

```
struct empleado {
  int codigo;
  float sueldo;
};

struct empleado emp,emp1;
```

Las estructuras también se pueden inicializar en la declaración, por ejemplo:

```
struct empleado {
  int codigo;
  float sueldo;
} emp = { 8652,1210.25 };
```

En las secciones siguientes se presentará el uso combinado de estructuras con otros objetos de datos estructurados que ofrece C, por ejemplo, los arreglos y las uniones.

6.3. Uniones

En una estructura, los distintos campos ocupan posiciones separadas de memoria. Por ejemplo:

```
struct empleado {
  int codigo;
  float sueldo;
} emp;
```

se representa en memoria como se muestra en la parte izquierda de la figura 1 [3].

Una unión es similar a una estructura, con la diferencia que todos los campos comparten el mismo lugar en memoria. Por ejemplo:

```
union valor {
  int valor_int;
  float valor_float;
} val;
```

se representa en memoria como se muestra en la parte derecha de la figura 1.

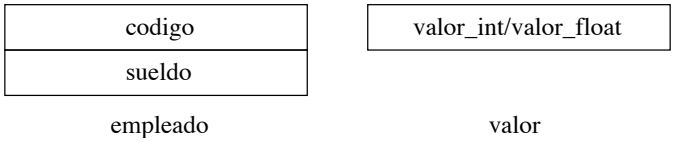

Figura 1: representación de estructuras y uniones

Los campos se acceden de igual forma que en una estructura, pero a diferencia de la estructura en la que los campos no tienen ninguna relación entre si, en una unión asignar un valor a uno de los campos destruye el valor que estaba asignado en los otros.

Pregunta 27 El siguiente programa utiliza la unión declarada anteriormente. Indique el efecto de cada asignación, indicando si los resultados que se obtienen son válidos.

```
int main(){
  int i;
```

[3] El tamaño de los campos puede ser distinto en memoria, pero no se ha tenido en cuenta en la figura

```
      float f;
      union valor {
        int valor_int;
        float valor_float;
      } val;

      val.valor_float = 2.1;
      val.valor_int = 4;
      i = val.valor_int;
      f = val.valor_float;
      val.valor_float = 3.3;
      i = val.valor_int;
      return(0);
    }
```

6.4. Combinación

Los arreglos, las estructuras y las uniones pueden ser combinadas. Por ejemplo, un arreglo de 10 estructuras `empleado` se define como sigue:

```
struct empleado {
  int codigo;
  float sueldo;
} emp[10];
```

Sus componentes se pueden acceder en la forma esperada:

```
emp[0].codigo = 8652;
emp[5].sueldo = 1210.25;
```

Se pueden inicializar componentes del arreglo de estructuras en la definición como sigue:

```
struct empleado b[3] = { { 1011,775.45 },
                         { 3441,1440.90 } };
```

Esta definición inicializa las dos primeras componentes del arreglo `b`.

Las estructuras también pueden contener arreglos. Por ejemplo:

```
struct {
  int productos[10];
  float monto_en_pesos;
  float monto_en_liras;
} prod;
```

Pregunta 28 ¿Cómo se accede a la cuarta componente del arreglo `productos` de la variable `prod`?

También se pueden combinar con uniones. El siguiente ejemplo permite almacenar informacion sobre productos nacionales e importados almacenando la información conveniente en cada caso:

```
struct {
  int codigo;
  struct {
    float precio_basico;
    float impuestos;
  } precio;
```

```
  int nacional;      /* 1 = nacional, otro valor = importado */
  union {
    int cod_prov;    /* codigo provincia si es nacional */
    long cod_pais;   /* codigo pais si es importado */
  } cod_origen;
  int existencia;    /* numero de productos en existencia */
} prod[50];
```

Por ejemplo, las papas código 423 producidas en San Luis se podrían almacenar asi:

```
prod[0].codigo = 423;
prod[0].precio.precio_basico = 0.23;
prod[0].precio.impuestos = 0.07;
prod[0].nacional = 1;
prod[0].cod_origen.cod_prov = 5700;
prod[0].existencia = 3000;
```

y las sardinas código 1077 de España asi:

```
prod[0].codigo = 1077;
prod[0].precio.precio_basico = 3.01;
prod[0].precio.impuestos = 0.72;
prod[0].nacional = 0;
prod[0].cod_origen.cod_pais = 56;
prod[0].existencia = 1000;
```

Pregunta 29 ¿Es posible utilizar el campo `cod_pais` para un producto nacional?

7. Funciones

7.1. Creación y utilización

Una función es un conjunto de declaraciones, definiciones, expresiones y sentencias que realizan una tarea específica.

Para la creación de una función, el formato general a seguir es el siguiente:

```
<especificador-de-tipo> <nombre-de-funcion> (lista-de-parametros)
{

    <variables locales a la funcion>
    <codigo de la funcion>

}
```

El **especificador-de-tipo** indica el tipo del valor que la función devolverá mediante el uso de la sentencia *return*. También puede especificar que no se devolverá ningún valor utilizando el tipo *void*. El valor, si fuera diferente a *void*, puede ser de cualquier tipo válido. Si no se especifica, entonces el procesador asume por defecto que la función devolverá un resultado entero.

No siempre se deben incluir parámetros en una función. En consecuencia, se asume que la **lista-de-parámetros** puede estar vacía.

Las funciones terminan y regresan automáticamente a la función invocante cuando se encuentra la última llave **}**, o bien, se puede forzar el regreso anticipado usando la sentencia *return*. (Esta sentencia permite, además, devolver un valor a la función invocante).

El siguiente ejemplo muestra el cálculo del promedio de dos números enteros mediante el uso de funciones:

```
float encontprom(int num1, int num2)
{
    float promedio;
    promedio = (num1 + num2) / 2.0;
    return(promedio);
}

int main()
{
    int a=7, b=10;
    float resultado;

    printf("Promedio=%f\n",encontprom(a, b));
    return(0);
}
```

7.2. Parámetros formales y Parámetros reales o actuales

En el ejemplo anterior la función **encontprom** emplea dos parámetros identificados como **num1** y **num2** ambos parámetros del tipo entero (*int*). Dichas variables se denominan **parámetros formales**.

Al igual que las variables definidas en el cuerpo de la función, los parámetros formales representan variables locales a la misma, cuyo uso está limitado a la propia función. Por lo tanto, no existe problema alguno al utilizar nombres duplicados de variables en otras funciones.

Es importante notar que el lemguaje C requiere que cada variable parámetro vaya precedida por su tipo. Será un encabezado no válido si se escribe:

```
float encontprom(int num1, num2) ***No válido***
```

En el ejemplo de la sección 7.1 cuando se invoca a la función **encontprom** en la función **main** las variables **a** y **b** representan los **parámetros reales o actuales**. Los contenidos de estas variables se copian a los correspondientes parámetros formales de **encontprom**, las variables **num1** y **num2**.

Así, un **parámetro formal** es una variable en la función que ha sido invocado, mientras que el **parámetro real o actual** puede ser una constante, variable o incluso una expresión más elaborada que forma parte de la invocación. Todo argumento real es evaluado y se envía su valor a la función.

Por ejemplo:

```
encontprom(a+5, b);
```

La expresión del primer parámetro real se calcula, obteniendo como resultado 12 (ya que el valor de a es 7) el cual se asigna a la variable **num1**. La función **encontprom** desconoce si el número procede de una constante, una variable o una expresión más general.

7.3. Tipos de una función

7.3.1. Devolución de un valor desde una función

A partir de los parámetros se puede comunicar información desde una función que invoca, a otra que recibe la llamada (en el ejemplo anterior, sería desde la función *main* a la función *encontprom*). Para enviar información en la dirección contraria, se utiliza el valor de "retorno" de la función. La palabra clave **return** hace que un valor se transmita como retorno de la función invocada, en el ejemplo el valor a retornar es el contenido de la variable **promedio**. Una función que posea valor de retorno deberá declararse con el mismo tipo que tiene dicho valor. En el ejemplo, el tipo de **promedio** es *float*, por lo tanto el tipo de la función *encontprom* también deberá ser *float*.

Como ya fue comentado anteriormente, si no se indica el tipo de una función el lenguaje supone que la función es de tipo *int*.

Es importante tener en cuenta que la declaración de tipo forma parte de la definición de la función, refiriéndose a su valor de retorno y no a sus parámetros, así en el encabezado siguiente:

```
float encontprom(int num1, int num2)
```

se está definiendo una función que acepta dos parámetros de tipo *int* pero devuelve un valor de tipo *float* (por lo tanto la función se dice que es de tipo *float*). Al momento de invocar a una función que retorna un valor es posible asignar a alguna variable el valor devuelto por la misma. Dicha variable deberá ser del mismo tipo que el de la función.

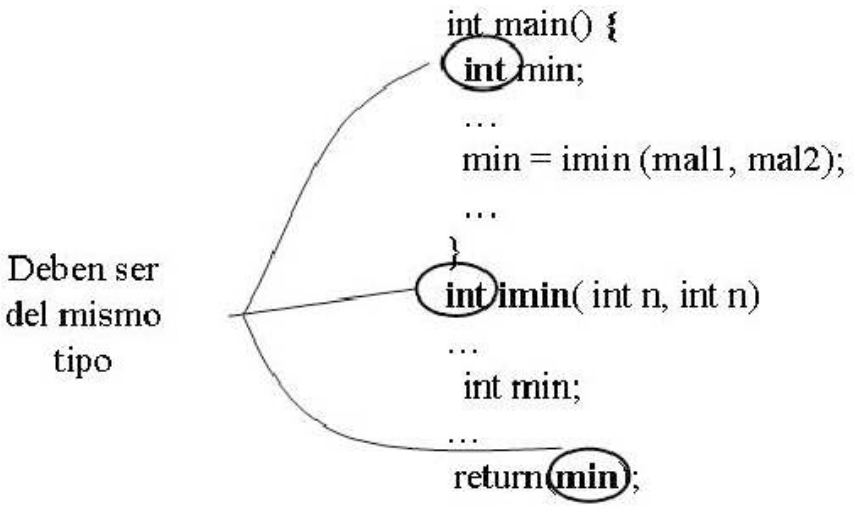

Figura 2: Ejemplo de devolución de un valor desde una función

7.3.2. Funciones de tipo *void*

Las funciones de tipo *void* se definen cuando no se requiere regresar un valor. La función en este caso realiza la tarea por la que se invocó pero no retorna ningún valor a la función que la llamó, por lo tanto no es necesaria

la utilización de la palabra clave "return". En la subsección siguiente se muestra un ejemplo con una función de tipo *void* cuya única tarea es la de imprimir el resultado obtenido al ejecutar otra función.

7.4. Un programa más complejo

A continuación se presenta un programa más elaborado conteniendo implementaciones de funciones de los dos tipos explicados en las subsecciones 7.3.1 y 7.3.2.

```
#include <stdio.h>

int doble(int x) {
  return (2 * x);
}

void mostrar(int x, int y){
  printf("el doble de %d es %d\n",x,y);
  return;
}

int main() {
  int result;

  result = doble(5);
  mostrar(5,result);

  return(0);
}
```

Este programa incluye dos funciones además de la función `main`. La función *doble* y la función *mostrar*

Dentro de la definición de la función `main`, aparece en la primera línea la declaración de una variable local llamada `result` de tipo entero. Notar que, las declaraciones de las variables locales aparecen dentro del cuerpo de la función (es decir dentro de las llaves). Esta variable, por ser local a la función `main`, sólo podrá ser utilizada dentro de ella.

La función *doble* se especifica a través de:

```
int doble(int x)
```

Esta especificación indica que la función tiene como nombre `doble`, retorna un valor entero y toma como argumento un entero a través del parámetro formal `x`.

La función *mostrar* se especifica a través de:

```
void mostrar(int x, int y)
```

Esta especificación indica que la función tiene como nombre *mostrar*, no retorna ningún valor y toma como argumento dos valores entero a través de los parámetros formales `x` e `y`.

La función `doble` es invocada desde el programa principal (o función `main`) a través de la sentencia:

```
result = doble(5);
```

En esta expresión se asigna a la variable `result` el valor entero retornado por la función `doble` al ser invocada con el valor 5. Tanto la variable *result* como la función *doble* son del tipo `int`.

La función `doble` es invocada, y el parámetro formal `x` toma el valor 5. La función `doble` tiene una única sentencia en su cuerpo:

```
return (2 * x);
```

El valor del parámetro formal `x` es multiplicado por 2 y ese resultado es retornado por la función.

Al retornar a la función `main`, el resultado de la invocación es asignado a la variable local `result`.

A continuación la función `mostrar` es invocada, y los parámetros formales `x` e `y` toman los valores de los parámetros reales 5 y el valor contenido en result, respectivamente. La función `mostrar` tiene como única sentencia en su cuerpo la de imprimir a través de la función `printf` de la siguiente forma:

```
printf("el doble de %d es %d\n",x,y);
```

obteniéndose en la pantalla:

```
el doble de 5 es 10
```

Notar que la función `printf` recibe tres parámetros, el segundo y tercero de ellos corresponden a los valores de los parámetros `x` e `y`. El primer parámetro de la función `printf` se conoce como *string de formato*, a través del cual es posible especificar la forma en la que se debe realizar la impresión. Los caracteres de este string son impresos tal cual como fue visto en el ejemplo anterior. El uso del caracter % indica el comienzo de una *secuencia de especificación de formato*. La secuencia no es impresa, sino que es reemplazada por el valor de la variable que aparece a continuación, el cual será impreso de acuerdo a las indicaciones del especificador. En el ejemplo, el primer especificador `%d` indica que el valor de la primer variable (segundo parámetro del "printf"), `x` en este caso, será impreso como un entero decimal. Esta es la razón por la que el valor 5 es impreso como parte de la oración.

Pregunta 30 En el string de formato de la función `printf` también se puede utilizar el especificador de formato `%x` para imprimir el valor de la variable en hexadecimal. ¿Cuál es la salida del siguiente trozo de programa?

```
#include <stdio.h>

int main() {
  int i;

  i = 255;
  printf("%d %x\n",i,i);
  return(0);
}
```

8. Punteros

Los punteros (o apuntadores) son una de las herramientas más poderosas de C, sin embargo en este caso, poder implica peligro. Es fácil cometer errores en el uso de punteros, y estos son los más difíciles de encontrar, una expresión típica usada en este caso por los programadores se refiere a la "pérdida de un puntero", lo cual indica que ocurrió un problema en la asignación de algún puntero. La ventaja de la utilización de punteros en C es que mejora enormemente la eficiencia de algunos procesos, además , permite la modificación de los parámetros pasados a las funciones (paso de parámetros por referencia) y son usados en los procesos de asignación dinámica de memoria.

Las variables están almacenadas en algún lugar de la memoria. Por ejemplo, si se define una variable entera como sigue:

```
int i = 10;
```

en algún lugar de la memoria, por ejemplo en la dirección 3000, se reserva un lugar para almacenar los valores de esta variable. La situación en memoria se muestra en la figura 3.

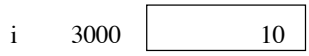

Figura 3: variable en memoria

Una variable puntero `p`, de tipo entero se define de la siguiente manera:

```
int *p;
```

y puede apuntar a la variable **i** asignándole la dirección de la variable. La dirección de un objeto de datos se puede obtener usando el operador **&**:

```
p = &i;
```

La situación en memoria se muestra en la figura 4 donde se puede apreciar que el puntero **p** contiene la dirección de la variable **i**, o lo que es lo mismo, apunta a la variable **i**.

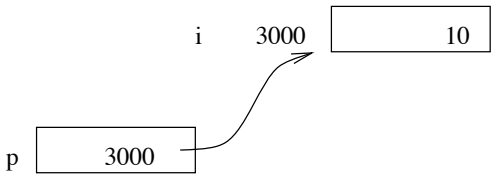

Figura 4: la variable es apuntada por el puntero

Se puede concluir que un puntero es una variable que contiene una dirección de memoria, es decir, contiene la dirección de un objeto o variable.

Los únicos operadores válidos en las variables puntero son:

Operador **&** y el operador $*$ que a continuación se describe.

`Operador &`

Contiene la dirección o posición de memoria en la cual se ha almacenado una variable. El operador **&** es unario, es decir, tiene un solo operando, y devuelve la dirección de memoria de dicho operando.
suponga el siguiente ejemplo:

```
int main()
{
int auxiliar = 5;
printf("\n auxiliar = %d ---> direccion = %p", auxiliar, &auxiliar);
return(0);
}
```

por pantalla saldrá auxiliar = 5 —> dirección=289926.

En este ejemplo se utilizó el modificador de formato **%p** que muestra la dirección.

Los punteros se declaran en los programas para después utilizarlos y poder tomar las direcciones como valores.

Por ejemplo:

```
int *punt; /* se declara un puntero a int */
int j = 5;
```

Se declaró un puntero **punt** a enteros, es decir, el contenido del puntero es una dirección que apunta a un objeto de tipo entero. Si se hiciese la siguiente asignación:

```
punt = &j;
```

estaría correcta e implicaría que **punt** apunta a la dirección de memoria de la variable **j**. Gráficamente se muestra en la figura 5:

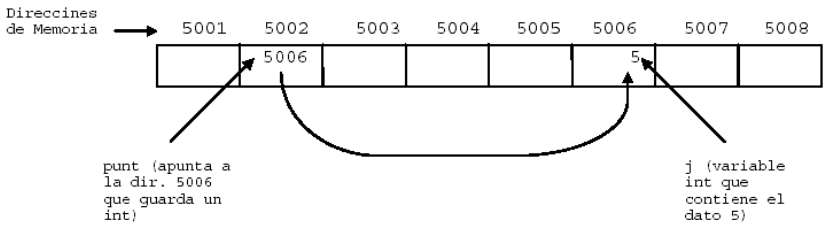

Figura 5: Punteros

Los punteros pueden ser inicializados, para ello existen varias formas:

```
punt = NULL;
punt = (int *) 285395; /* asignacion de direccion */
/* absoluta */
```

En la primera línea se le asigna un nulo, es como decir que todavía no apunta a ningún elemento. En la segunda línea se hace la asignación de una posición específica de memoria.

Entonces, para declarar un puntero, primero se especifica el tipo de dato (básico o creado por el usuario), luego el nombre del puntero precedido por el carácter *.

Operador $*$

El operador * es unario y toma a su operando como una dirección de memoria, entonces el operador accede al contenido de esa dirección. Se llama también operador de desreferenciación y nos permite acceder al valor apuntado por un puntero.

Por ejemplo, suponga las variables declaradas anteriormente y la asignación,

```
int *punt;
int j = 5;
punt = &j;
printf("El contenido de punt es: %d",*punt);
```

en pantalla imprimiría:

El contenido de punt es: 5

Es decir imprimiría el contenido de la dirección a la que apunta **punt**. Habría sido igual haber impreso la variable **j**. Si se declarara otra variable **int**, igualmente es válido hacer:

```
int *punt;
int j =5 , otro;
punt = &j;
:
:
otro = *punt;
```

que asignaría a otro el dato 5. Si se hace la siguiente asignación:

```
*punt = 200;
```

se modifica el contenido de la dirección a la que apunta **punt**. Por lo tanto también se modificó la variable j, ya que la dirección de esta es a la que apunta **punt**.

Pregunta: cuál es el contenido de la variable **otro** después de la última asignación?

Hay ciertas restricciones en la asignación de punteros, por ejemplo:

```
int *p,*q,*r; /* 3 punteros a entero*/
int array[20], var1;
q = 568200;
r = &array;
p = &(var1 + 5);
```

Todas las asignaciones anteriores son **erróneas**. A un puntero no se le puede asignar un valor cualquiera si no existe una conversión explícita de tipo. No puede recibir la dirección de un nombre de un arreglo dado que este último es un puntero al primer elemento del arreglo. La última asignación también es ilegal ya que se pretende asignar la posición de **var1** mas 5 posiciones de memoria. Las siguientes asignaciones son correctas:

```
int *p,*q;
int array[20],var1;
p = NULL;
q = &array[5];
p = q;
```

Primero a **p** se le asignó el valor nulo, luego a **q** se le asignó la dirección del sexto elemento del arreglo y finalmente a **p** se le asignó la misma dirección que **q**, por lo tanto apuntan a la misma posición.

Pregunta 31 Suponga que se declaran dos punteros y una variable entera, y se hace apuntar a ambos punteros a la misma variable entera:

```
int *p,*q;
int i = 5;

p = &i;
q = &i;
```

¿Qué valores imprime el siguiente trozo de programa?

```
*p = *p + 1;
printf("%d %d %d\n",*p,*q,i);
i++;
printf("%d %d %d\n",*p,*q,i);
```

Hasta ahora se ha usado implícitamente punteros en nuestros programas, por ejemplo al invocar la función **scanf**, donde se pasa como parámetro la dirección de la variable, o cuando al invocar una función con arreglos como parámetro actual, modifica los valores contenidos.

8.1. Punteros como parámetros

En C el pasaje de parámetros es por valor, por lo que se torna imposible la definición de funciones que modifiquen el contenido de los parámetros actuales de una invocación. Se sabe que cuando se pasa como parámetro una variable a una función, esta no logra modificar el valor de la variable, esto ocurre porque fue

pasada por valor. Por ejemplo, en el siguiente programa se define una función que tiene como objetivo (no logrado) incrementar el valor de la variable que es pasada como argumento [4].

```
#include <stdio.h>

void incremento(int x) {
  x++;
  return;
}

int main()
{
  int i = 0;
  while (i < 5) {
    printf("%d",i);
    incremento(i);
  }
  return(0);
}
```

El programa no termina nunca su ejecución dado que no logra incrementar el valor de la variable que es pasada como argumento. El parámetro formal **x** es una **copia** del valor de **i**, y por lo tanto, si bien es incrementado dentro del cuerpo de la función **incremento**, el valor de **i** no es modificado (ver figura 6).

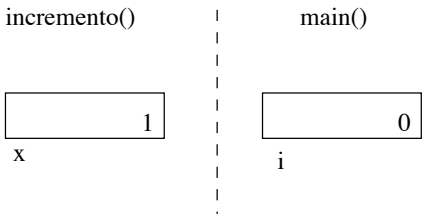

Figura 6: pasaje por valor

Para lograr que se modifique el contenido de una variable debe pasarse por referencia, para ello se pasa a la función la dirección de la variable y no el dato. El programa modificado se presenta a continuación:

```
#include <stdio.h>

void incremento(int *x) {
  (*x)++;
  return;
}

int main() {
  int i = 0;

  while (i < 5) {
    printf("%d",i);
    incremento(&i);
  }
```

[4]El tipo de valor retornado por la función se define como **void**, esto significa que en realidad no devuelve ningún valor

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

Observar que se pasa la dirección de la variable, y el parámetro formal se define como un puntero, ya que éste recibirá una dirección. Notar además como las referencias al parámetro formal en la función **incremento** han debido ser modificadas. La descripción de la situación en la memoria se presenta en la figura 7.

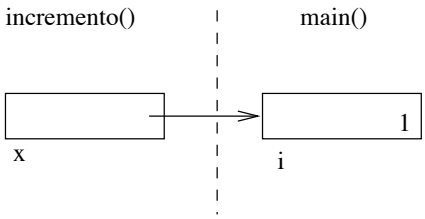

Figura 7: pasaje por valor de una dirección

Suponga que se desea hacer una función que haga el intercambio de datos entre dos variables (swap).

Solución: Primero se hace el programa principal, de tal manera de saber como se hace la llamada por referencia. Este podría ser:

```
#include <stdio.h>


int main()
{
  int a = 2 , b = 4;

  swap(&a,&b);
  return(0);
}
```

El programa resulta bastante simple, lo importante es notar que ahora los parámetros a y b van precedidos del operador **&**, es decir, no se pasan los datos 2 y 4, sino, las direcciones de las variables. Dado que como la función recibió las direcciones de variables int, se deberá declarar en la cabecera de la función parámetros formales del tipo punteros a int. Esto sería: void swap(int *x,int *y).

Dentro de la función lo que se debe hacer es intercambiar los valores. Como los parámetros son punteros, entonces se debe trabajar con los contenidos de dichos punteros (o los contenidos de las direcciones a la que apuntan los punteros), para ello utilizamos el operador *. La función completa utilizando punteros es:

```
void swap(int *x,int *y)
{
  int temp = *x;
  *x = *y;
  *y = temp;
  return;
}
```

Con esto se logra el objetivo, la salida al programa es:
a=2 b=4

a=4 b=2

La figura 8 muestra la evolución de los valores durante la ejecución de la función **swap**.

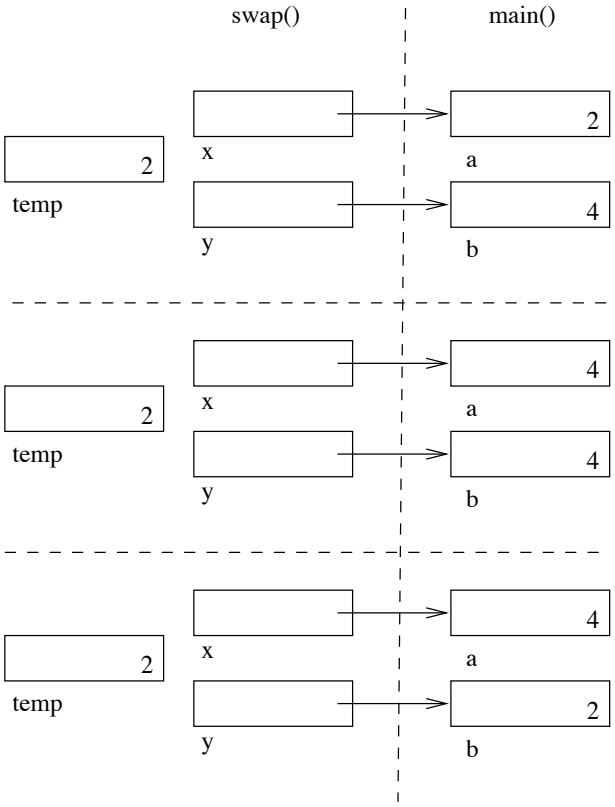

Figura 8: pasos de la ejecución de la función swap

Ejercicio: Crear un programa que lea dos números enteros, luego una función que reciba como parámetros los números ingresados y mediante una invocación a la función **swap** ya creada, permitir que salga el mayor de los números en el primer parámetro y el menor de los números en el segundo.

Solución: El programa principal sería:

```
#include <stdio.h>
void maxymin (int *x, int *y);
int main(){
  int i,j;

  scanf("%d",&i);
  scanf("%d",&j);
  maxymin(&i,&j);
  printf("El Max es i=%d, El Min es j=%d",i,j);
  return(0);
}
```

La función no variará mucho respecto al ejercicio anterior:

```
void maxymin(int *x,int *y){
 int aux;
   if (*x<*y){
     aux = *x;
     *x = *y;
     *y = aux;
   }
   return;
}
```

En la sentencia **if** se preguntó si "el contenido del puntero x es menor que el contenido del puntero y". En esta función se utiliza la función anterior **swap** para hacer el cambio. En ese caso, aparte de incluir la declaración de dicha función antes del **main**, **maxymin** queda:

```
void maxymin(int *x,int *y){
  int aux;
  if (*x<*y)
    swap(x,y);
  return;
}
```

Hay que notar que tanto **x** como **y** son variables punteros. Esto quiere decir que **swap** recibe como parámetros las direcciones almacenadas en **x** e **y**; los datos ***x** y ***y** serán efectivamente intercambiados dentro de la función **swap**.

8.2. Punteros y arreglos

Los punteros y los arreglos están muy relacionados en C. Todas las operaciones que se pueden realizar con arreglos se pueden realizar también usando punteros. Las siguientes instrucciones definen un arreglo de enteros y un puntero a enteros:

```
int a[5] = { 7,4,9,11,8 };
int *p;
```

Luego de la asignación:

```
p = &a[0];
```

Como ya se dijo en C es posible realizar una gran cantidad de operaciones con punteros.
Por ejemplo, es posible sumar un entero a un puntero que apunta a una componente de un arreglo. Por definición, si el puntero **p** apunta a una componente de un arreglo, **p+i** apuntará **i** componentes más adelante (vea la figura 9).

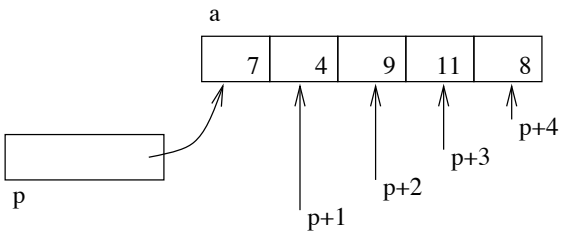

Figura 9: sumando constantes a un puntero

Es decir, que por ejemplo, el valor de la cuarta componente del arreglo **a** se puede asignar a la variable **x** a través del índice:

```
x = a[3];
```

o a través del puntero:

```
x = *(p+3);
```

En general, si un puntero **p** apunta al comienzo de un arreglo **a**, entonces ***(p+i)** es equivalente a **a[i]**.

Pregunta 32 Dadas las siguientes definiciones y asignaciones:

```
int a[3] = { 5,8,1 };
int *p,*q;
int i;

p = a;
q = &a[1];
```

Grafique el arreglo y los lugares a los que apuntan los punteros, e indique el efecto de cada una de las siguientes operaciones:

```
i = *p + *q;
*p += *q;  /* o lo que es lo mismo: *p = *p + *q; */
*(p+1) = 0;
(*(q+1))++;
```

También se puede pasar arreglos como parámetros. Considere el siguiente ejemplo, en el que se invoca una función **f** con el nombre del arreglo como parámetro:

```
int a[5];
f(a);
```

El nombre del arreglo por si mismo representa la dirección base del arreglo, por lo que se pasa un puntero a enteros que apunta a la primera componente del arreglo. Es decir, que la función **f** se puede definir con un argumento de tipo puntero a enteros:

```
void f(int *p) {

  *(p+1) = 3;
  return;
}
```

La asignación ***(p+1) = 3** asignará el valor 3 a la segunda componente del arreglo **a** en forma indirecta a través del puntero **p**.

Sin embargo, también se puede definir el argumento como un arreglo de enteros sin especificar el tamaño:

```
void f(int x[]) {

  x[1] = 3;
  return;
}
```

y la asignación **x[1] = 3** tiene el mismo efecto que la anterior.

En C, ambas declaraciones son equivalentes independientemente de como se declare el argumento(*int *p* ó *int x[]*). Esto significa que la función se puede codificar de cualquiera de las siguientes maneras:

```
void f(int *p) {            void f(int x[]) {

  *(p+1) = 3;                *(x+1) = 3; // usando notacion puntero
  p[2] = 4;  //usando notacion arreglo  x[2] = 4;
  return; return;
}                           }
```

Suponga el siguiente ejemplo:

```
int *p,arre[20],i;
:
:
p=&arre[0];
for( i = 0; i < 20; i++)
{
  printf("%d",*p);
  p++;
}
```

En este caso se imprime todo el arreglo *arre* utilizando el puntero **p**. Esto es efectivo dado que al declarar el arreglo, sus elementos se ubican en posiciones contiguas de memoria. También sería correcto utilizar

```
p = &arre[0];
printf("El decimo elemento es : %d",*(p+9));
```

para hacer referencia al décimo elemento del arreglo. Como hay que tener cierto cuidado con el manejo de punteros, veamos este ejemplo:

```
int *p, arreglo[20], i;
:
:
p = &arreglo[0];
for(i = 0; i < 20; i++){
  printf("%d",*p);
  p++;
}
printf("El decimo elemento es : %d",*(p+9));
```

Este es similar al anterior, sin embargo, no hace lo que supuestamente se desea, porqué?, la razón es que el puntero fue incrementado 20 veces y después del for ya no apunta al primer elemento del arreglo, por lo tanto puede imprimir cualquier valor (que sería aquel ubicado 9 variables enteras más allá). También es posible hacer la resta entre dos punteros, esto con el objeto de saber cuantos datos del tipo base (de los punteros) se encuentran entre ambos. Igualmente se puede hacer la comparación entre punteros:

```
if ( p > q )
  printf("\n p es mayor que q\n");
```

En este caso indicaría que **p** está ubicado en una posición de memoria superior a la de **q**.

Pregunta 33 ¿Es posible realizar operaciones tal como ***(a+i) = 6** si se asume que **a** es un arreglo de enteros?. En caso afirmativo, ¿Cuál es su significado?.

Pregunta 34 ¿Cuál es el objetivo de la siguiente función?

```
void f(int n,int x[]) {
  int i;

  for(i = 0;i < n;i++) x[i] = 0;
  return;
}

int main() {
  int a[10],b[20];

  f(10,a);
  f(20,b);
  return(0);
}
```

Pregunta 35 Considere la misma función **f** de la pregunta anterior. ¿Cuál es el efecto de las invocaciones?

```
f(5,a);
f(10,&b[5]);
```

.

Algo más sobre punteros

Los punteros no solamente pueden apuntar a tipos conocidos, sino que también a tipos creados por el usuario, como es el caso de punteros a estructuras. Otro ejemplo de uso de punteros es cuando un arreglo es definido como un arreglo de punteros, por ejemplo, la siguiente declaración crea un arreglo de punteros de 5 elementos:

```
int *arrepunt[5];
```

donde cada una de las componentes del arreglo es un puntero a entero. En consecuencia, sentencias como las siguientes son correctas:

```
arrepunt[0]=&var1;
printf("\n%d",*arrepunt[0]);
```

donde, al primer elemento del arreglo se le asignó la dirección de la variable entera **var1**, posteriormente se imprimió el contenido de dicho puntero.

Un ejercicio gráfico

El ejercicio de la figura 10 pretende mostrar gráficamente que ocurre con las variables y punteros durante la creación y asignación.

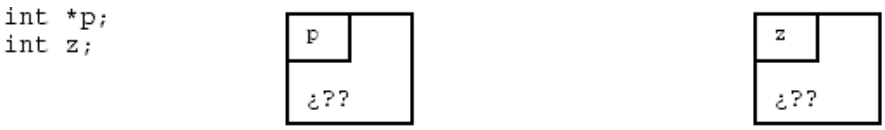

Figura 10: Creación

Después de la declaración, se crea una variable entera **z** la cual tiene su espacio en memoria, aunque su valor es desconocido (basura). Además, una variable puntero **p (a int)** la cual aún no tiene espacio asignado (para contener valores) y no apunta a ningura parte.

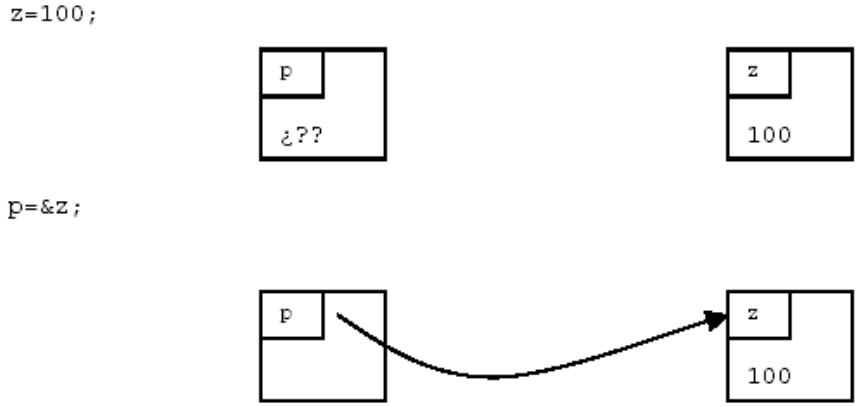

Figura 11: Asignación

Despues de las asignaciones de la figura 11, la variable **z** contiene el valor 100 y la variable **p** apunta a la dirección donde está **z**. Recién ahora se puede hablar del contenido de **p**, es decir, si se ejecuta:

```
printf("%d",*p);
```

se imprime el valor 100.

Suponga ahora que se hace la siguiente asignación, ver figura 12,

donde NULL es la forma de representar que el puntero apunta a nada. En la figura se muestra como barras paralelas decreciendo en tamaño. **p** ha dejado de apuntar a la variable **z**, siendo ahora su valor NULL. Si luego se hace:

```
p=(int *)malloc(sizeof(int));
```

se crea el espacio necesario en memoria para guardar un valor entero, a la que apuntará la variable **p**, a este valor sólo se puede accesar a través de **p**.

Gráficamente se muestra en la figura 13:

Después de esta asignación, el contenido de **p** es igual a 150, y **z** aún mantiene el valor 100.

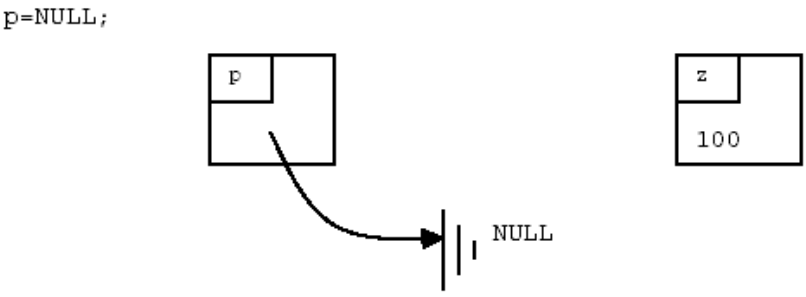

Figura 12: Valor NULL

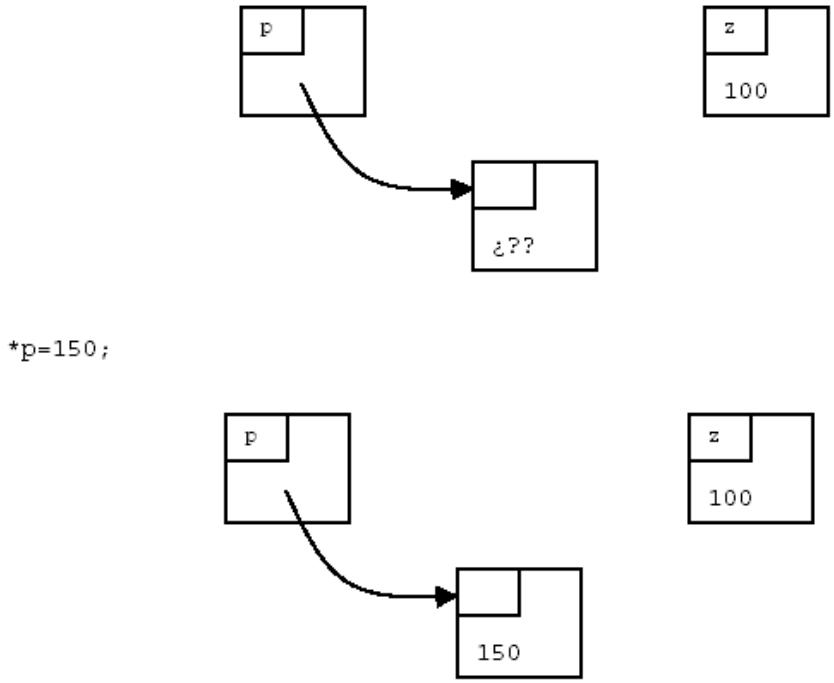

Figura 13: Reserva de memoria y asignación de valor

Si ahora se hace nuevamente (ver figura 14):

Con esto, en la jerga se dirá que: "se perdió un puntero". En realidad, formalmente **p** no está perdido, si no que ahora apunta a **z**, pero nos es imposible recuperar el valor que tenía antes. Como se muestra en la figura, no hay ninguna forma de acceder al espacio de memoria que contiene el valor 150, siendo que ese espacio sigue ocupado. Si después de esto, se ejecuta la siguiente línea:

```
*p = 200;
```

ocurre lo que se muestra en la figura 15.
Ahora, tanto **z** como el contenido de **p** tienen el valor 200. Finalmente, si:

```
printf(%d %d,z,*p);
```

imprimiría en pantalla:

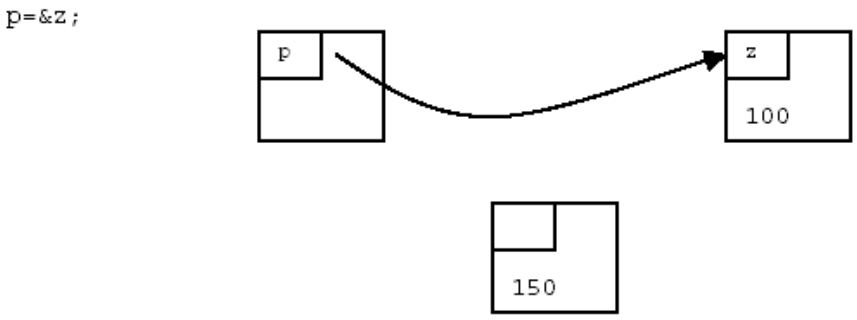

Figura 14: Uso del espacio de memoria

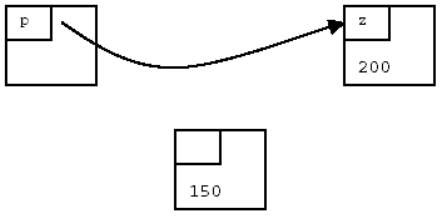

Figura 15: Uso del espacio de memoria

200 200

Un último comentario es recalcar que lo que almacena **p**, es una dirección de memoria, es decir, la dirección de memoria a la que está apuntando, en los esquemas anteriores ésta se representa a través de la flecha.

Problemas Propuestos

1.- Para el siguiente extracto de programa:

```
int *p, arreglo[6], i;
:
: /* datos del arreglo: 5 4 56 32 21 50 */
p = arreglo;
for( i = 0; i < 6; i++)
{
printf("%d",*p);
printf("%p",p);
p++;
}
```

a) Indique que hace el programa.

b) Si el primer elemento del arreglo esta ubicado en la posición de memoria 13500, entonces, a que dirección de memoria queda apuntando p al finalizar el ciclo for?, escriba cada valor que va saliendo por pantalla.

2.- Se tiene la siguiente declaración:

```
long *p, array[20];
long m, n, otro[6];
```

En un momento determinado las posiciones de memoria para las variables son:

```
array comienza en 14560
m comienza en 14600
n comienza en 15000
otro comienza en 1510
```

Entonces, si se realiza la siguiente asignación:

```
p = array;
```

y a p se le incrementa en 4, luego en 17, en 3 y en -2.

Indique las posiciones para cada uno de los incrementos.

3.- Escriba un programa que lea 100 números enteros y los almacene en un arreglo. Luego escriba una función que entregue a la función principal el valor máximo y mínimo, para ello ocupe parámetros pasados por referencia (a la función se le pasan tres parámetros: el arreglo y dos pasados por referencia que contendrán el máximo y mínimo).

4.- Escriba un programa que ingrese primero 2 datos (x,y) y que calcule la potencia de x elevado a y, luego un tercer dato (z) para el cual se calculará el factorial. La potencia y el factorial deben ser funciones a las cuales se le pasan parámetros por referencia. Los datos (x,y,z) deben ser leídos desde el programa principal.

8.3. Aritmética de punteros

Las operaciones matemáticas que se pueden usar con punteros son el decremento, el incremento y la resta entre dos punteros, el resto de las operaciones quedan prohibidas. La operatoria de punteros queda sujeta al tipo base de este, por ejemplo:

```
int *p,*q;
int dato1;
int dato2[10];
p = &dato1;
q = dato2;
q++;                    //valido, incrementa el puntero una posicion del arreglo
q = q + 5;               //valido, incrementa el puntero cinco posiciones del arreglo
q = q - 2;               //valido, decrementa el puntero dos posiciones del arreglo
q = q - p;               //valido, obtiene la diferencia en bytes entre las dos direcciones
q = p + p;               //invalido, no se pueden sumar dos direcciones
q = q * 5;               //invalido, no se puede multiplicar un puntero
```

8.4. Punteros a estructuras

C admite los punteros a estructuras del mismo modo que admite punteros a cualquier otro tipo de variable. Sin embargo, hay algunos aspectos especiales de los punteros a estructuras, los cuales se detallarán a continuación.

8.4.1. Declaración de un puntero a estructuras

Los punteros a estruturas se declaran colocando el * delante del nombre de una variable estructura. Por ejemplo, si:

```
struct empleado{
        int codigo;
        char nombre[20];
        float sueldo;
}
```

es la definición global de un tipo estructurado llamado .empleado", entonces la declaración de una variable estructura es:

```
struct empleado emp; /* declara una variable de tipo struct empleado */
```

y la declaración de una variable puntero llamada "p_emp"de tipo "struct empleado"sería:

```
struct empleado *p_emp; /* declara un puntero a estructura empleado */
```

Para encontrar la dirección de una variable estructura, se utiliza el operador **"&"** delante del nombre de la variable estructura. Por ejemplo:

```
p_emp = &emp; /* "p_emp" se inicializa con la direccin de "emp" */
```

Para acceder a los elementos de la estructura, se usa el operador **"*"**. Ejemplo, para acceder al campo çodigo"se escribe:

```
(*p_emp).codigo
```

Como es muy común acceder a un campo de una estructura a través de un puntero, se ha definido un operador especial en C para llevar a cabo esta tarea. Se trata del operador "→". El operador flecha se utiliza en lugar del operador punto cuando se accede a un elemento de una estructura empleando un puntero a la variable estructura. Por ejemplo, la sentencia anterior se puede escribir de la siguiente manera:

```
p_emp -> codigo
```

8.4.2. Utilización de punteros a estructuras

Los punteros a estructuras tienen varias aplicaciones. Una de ellas es conseguir el pasaje de la dirección de una variable de tipo estructura a una función.

Pasar a una función un puntero a estructura tiene la ventaja de hacer una invocación muy rápida a la función. Tener en cuenta que siempre es más eficiente pasar como parámetro los cuatro bytes de un puntero, que pasar como parámetro la cantidad de bytes que ocupen todos los campos implicados en la variable estructura que se desea pasar a la función. Además, como la función estará haciendo alusión a la estructura en sí, y no a una copia, se podrá modificar el contenido de los elementos de la estructura utilizada en la invocación.

Ejemplo:

```
#include <stdio.h>

struct empleado{          //definicion "global" del tipo empleado
       int codigo;
       char nombre[20];
       float sueldo;
};

int ingresar (struct empleado *a){    //se recibe un puntero al tipo struct empleado
    scanf ("%d", &((*a).codigo));     //acceso al campo codigo mediante puntero
    getchar();
    scanf ("%[A-Z a-z]", a->nombre);  //acceso al campo nombre mediante puntero
```

```
    getchar();
    scanf("%f", &(a->sueldo));        //acceso al campo sueldo mediante puntero
    getchar();
    return (0);
}

int main(){
    struct empleado un_emp;
    ingresar (&un_emp);               //se pasa la "direccion" de la variable estructura "un_emp"
    return (0);
}
```

También es posible el pasaje de la dirección de solamente un campo de una variable de tipo estructura a una función.

Siguiendo el ejemplo anterior, se podría modificar el campo sueldo de la variable un_emp en una función, de la siguiente manera:

```
int modif_sueldo (float *s){
        printf ("Ingrese nuevo sueldo: \n");
        scanf ("%f", s);
        getchar();
        return (0);
}
```

donde s es un puntero con la dirección real del campo sueldo de la variable un_emp. La invocación a dicha función sería:

```
modif_sueldo (&(un_emp.sueldo));
```

9. Strings

Los strings son arreglos de caracteres. El caracter nulo, cuyo código ASCII es 0, se representa por `'\0'` y se utiliza para denotar el fin de un string. Por ejemplo, para construir el string `"hola"`, lo podemos hacer de la siguiente manera:

```
char str[5];

str[0] = 'h';
str[1] = 'o';
str[2] = 'l';
str[3] = 'a';
str[4] = '\0';
printf("string = %s\n",str); /* imprime: string = hola */
```

La secuencia de asignaciones crea el string de cuatro caracteres. Notar que es necesario el caracter adicional `'\0'`, denominado **terminador de string**, o **caracter nulo**, para indicar el fin de string. En el ejemplo el caracter `'\0'` lo coloca el programador.

La instrucción **printf**, utilizando el formato **" %s"**, imprime caracter a caracter hasta encontrar el caracter `'\0'`. La instrucción **scanf** utilizando el formato **" %s"**, lee caracter a caracter e incorpora al final de la cadena **en forma automática** el caracter `'\0'`.

Una notación más compacta para inicializar un string es la siguiente:

```
char str[5] = "hola";
```

que tiene exactamente el mismo efecto que la anterior. Tener en cuenta que el arreglo **str** se declaró de cinco elementos, un lugar más para albergar el caracter nulo.

La existencia de un caracter nulo para indicar el fin del string se debe a que los strings tienen longitud variable, pudiendo ser su longitud menor que la del arreglo base, por ejemplo:

```
char str[100] = "hola";           //str inicializado con una cadena constante,
                                  // de cuatro caracteres mas el terminador de string: '\0'
```

El nombre de un arreglo es además un puntero a la dirección base del arreglo (recuerde la sección 8.2). La siguiente declaración es válida:

```
char str1[] = "hola";              //str1 es un puntero constante
```

9.1. Funciones de strings de biblioteca en Linux

C admite una amplia variedad de funciones para la manipulación de cadenas de caracteres. Dichas funciones utilizan el mismo archivo de encabezado denominado **string.h**. Las funciones más comunes son:

- **strcpy()**

La función **strcpy** se utiliza para copiar el contenido de una cadena en otra. La forma general de la función **strcpy** es:

```
char *strcpy(char *s1, const char *s2)
```

El contenido de la cadena **s2** se copia en **s1**, incluyendo el caracter `'\0'`. La cadena **s1** debe ser suficientemente larga para almacenar en ella la cadena contenida en **s2**.

Ejemplo:

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

int main (){
char s1[20];
char s2[10]="copia";
strcpy(s1,s2);
printf("s1=%s y s2=%s \n",s1,s2);
return (0);
}
```

- **strcat()**

La función **strcat** añade **s2** a **s1**, incluyendo el caracter `'\0'`. Ambas cadenas deben estar terminadas con el caracter `'\0'`, y la cadena resultante contiene un `'\0'` al final. El caracter inicial de **s2** sobreescribe el caracter `'\0'` al final de **s1**. La cadena **s1** debe tener suficiente espacio para la cadena resultante.

La forma general de **strcat** es:

```
char *strcat(char *s1, const char *s2)
```

Ejemplo:

```
char s1[20];             //s1 debe tener suficiente espacio para la cadena resultante
char s2[10];
strcpy(s1, "hola,");
strcpy(s2, " amigos");
strcat (s1, s2);
printf("%s",s1);
```

- **strcmp()**

La función **strcmp** compara dos cadenas de caracteres, y retorna un cero si son iguales, sino retorna un número distinto de cero. La forma general de **strcmp** es:

```
int strcmp(const char *s1, const char *s2)
```

Ejemplo:

```
char s1[15]= "ls en linux";
char s2[20]= "LS en linux";
int i;

i= strcmp(s1,s2);         //el entero resultante se almacena en i, 0 si las cadenas con iguales
if (i!=0)
   printf("%s no es igual que %s \n", s1, s2);
else
   printf("%s es igual que %s \n", s1, s2);
```

- **strlen()**

La función **strlen** proporciona la longitud de una cadena de caracteres **cad**, sin incluir el caracter `'\0'`. La forma general de **strlen** es:

```
int strlen(const char *cad);
```

Ejemplo:

```
char cad[80];
printf("Introduzca una cadena: \n");
scanf("%s ", cad);
printf("%d", strlen(cad));
```

9.2. Funciones de strings definidas por el programador

El programador puede necesitar definir funciones para strings que no se encuentran el la biblioteca de C que ofrece Linux. Al momento de programar, es importante tener en cuenta que un string debe terminar siempre con el caracter `'\0'`.

El siguiente código ejemplifica como sería la invocación y definición de una función que calcula cuantos caracteres tiene un string. La función se llamará **longitud** y tomará el string como argumento (es decir, su dirección base) y retornará el número de caracteres que componen el string sin incluir el caracter nulo:

```
//codigo de invocacion a funcion longitud
char str1[10] = "abc";      // '\0' incluido en forma automatica
char str2[10];
str2[0] = 'h';
str2[1] = 'o';
str2[2] = 'l';
str2[3] = 'a';
str2[4] = '\0';            // '\0' incluido por programa

printf("%d \n", longitud(str1));       //imprime 3
printf("%d \n", longitud(str2));       //imprime 4
```

```
//codigo de la funcion longitud
int longitud (char s[]){
int i;

for (i=0; s[i]!= '\0'; i++);

return (i);
}
```

Notar que el programador asignó en **str2**, en la última posición el caracter nulo. De no ser así la función **longitud** debería recibir como parámetro el tamaño de la cadena.

La función **longitud** incrementa un contador `i`, y mientras en la posición `i` no aparezca el caracter nulo, continúa incrementándolo.

Notar además que no está permitido asignar usando la notación de strings constantes, es decir, que no es válido hacer:

```
char str[5];

str = "hola";  /* INCORRECTO */
```

Esto se debe a que si la asignación fuera válida, se permitiría cambiar la dirección base del arreglo `str`, que debería apuntar ahora a la dirección en la que se encuentran los caracteres.

Para poder asignar un string en otro, se deben copiar todos los caracteres, uno por uno. Para ello, se puede utilizar por ejemplo la función de biblioteca `strcpy`.

10. Entrada y salida estándar

Un programa C toma sus datos de la entrada estándar (stdin) e imprime en la salida estándar (stdout) . Por defecto, la entrada estándar es el teclado, y la salida estándar es la pantalla.

Para poder utilizar la entrada y salida estándar se necesitan funciones que se encuentran incluidas en la biblioteca estándar <stdio.h>.

La Entrada y Salida estándar se pueden utilizar de la siguiente manera:

10.1. Entrada y Salida Básica

Las funciones más comunes que permiten leer y escribir un caracter son getchar() y putchar(). Dichas funciones no hacen interpretación de la información ingresada o mostrada.

Se definen y se utilizan de la siguiente forma:

```
int getchar(void) -- lee un caracter de stdin. El valor leido debe ser luego
                     almacenado en una variable entera.
int putchar(char ch) -- escribe un caracter a stdout  el cual se encuentra
                        almacenado en ch.
```

Ejemplo:

```
int main() {
      int ch;

      ch = getchar();
      putchar(ch);
      return(0);
    }
```

10.2. Entrada y Salida Formateada

Las funciones más comunes que permiten leer y escribir una cadena de caracteres son scanf() y printf().

scanf

El prototipo de la función scanf está definido como:

```
int scanf(const char *formato, lista arg ...);
```

El scanf recibe como argumento un conjunto de parámetros donde el primero de ellos es una cadena con el formato de interpretación a otorgar a la información, y luego una lista de variables las cuales van a recibir los valores que deben coincidir con los del formato.

Ejemplo:

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

En este caso la lectura de la información debe interpretarse en formato decimal y el símbolo `'&'` que precede a la variable `a` es para especificar que lo que se está enviando como argumento no es el valor que tiene la variable `a` sino la dirección en donde debe almacenarse la información. Una vez efectuada la lectura, la información ingresada e interpretada como valor decimal es almacenada en la variable `a`.

Ejemplo:

```
scanf("%d %c", &num, &letra);
```

En este ejemplo, el orden de los argumentos es muy importante. El `"%d"` representa el formato decimal para la variable num y el `"%c"` indica el formato caracter para la variable letra.

En el caso de un arreglo o cadena sólo se requiere su nombre para poder usar `scanf`, ya que el mismo corresponde al inicio de la dirección de la cadena.

Ejemplo 1:

```
char cadena[80];
scanf("%s",cadena);
```

Ejemplo 2:

```
char letrasMay[80];
char vocales[20];
scanf("%[aeiou]",vocales); /* acepta unicamente las vocales minusculas */
scanf("%[A-Z]",letrasMay); /* acepta unicamente los letras en el rango de la A a la Z*/
```

printf

Esta función permite sacar por pantalla textos arbitrarios o el resultado de alguna operación.

El prototipo de la función está definido como:

```
int printf(const char *formato, lista arg ...);
```

El printf recibe como argumento un conjunto de parámetros donde el primero de ellos es una cadena con el formato de interpretación a otorgar a la información que va a ser mostrada, y luego una lista de variables las cuales contienen los valores a ser mostrados y cuyo tipo debe coincidir con los del formato especificado.

Ejemplo

```
int resultado;
printf ("Hola Mundo"); /* Imprime la cadena constante Hola Mundo */
resultado = 5+2;
printf ("%d", resultado); /* Imprime el contenido de la variable resultado, 7 */
```

En el siguiente ejemplo se muestra la importancia del orden y la correspondencia de los argumentos de la función `printf`.

Ejemplo

```
int num;
num = 3;
printf("El doble de %d es %d y su cuadrado es %d\n", num, num*2, num*num);
```

La salida del ejemplo sería: *El doble de 3 es 6 y su cuadrado es 9.*

Estas funciones, tanto `scanf` como `printf`, hacen "interpretación"de la información leída o mostrada. Por ejemplo, si se ingresa la cadena 123 en formato decimal se almacena como un único número (el 123), en cambio si lo ingresamos en formato string se almacena como una cadena de caracteres formada por el caracter 1, el caracter 2 y el caracter 3.

Los formatos más comunes utilizados tanto por `scanf` como por `printf` se encuentran en el Apéendice B.

Tanto la entrada estándar como la salida estándar se pueden cambiar. Una posibilidad es hacerlo a través de la redirección en la línea de comandos cuando se invoca el programa. Por ejemplo:

```
$ programa
$ programa < a.txt
$ programa > b.txt
$ programa < a.txt > b.txt
```

En el primer caso, la entrada y la salida son el teclado y la pantalla respectivamente. En el segundo, la entrada es redireccionada desde el archivo `a.txt`, esto quiere decir que, por ejemplo, la función `getchar` leerá caracteres de este archivo. En el tercer caso, la salida es redireccionada al archivo `b.txt`, por lo que, por ejemplo, la función `printf` imprimirá en el archivo `b.txt`. En el último caso, tanto la entrada como la salida han sido redireccionadas: la entrada desde el archivo `a.txt` y la salida hacia el archivo `b.txt`.

El punto importante aquí, es que las mismas funciones de entrada−salida son usadas, sin importar el dispositivo al cual han sido redireccionadas. Desde el punto de vista del programador, siempre se lee de la entrada estándar, y se imprime en la salida estándar.

En las secciones siguientes se presentarán sólo algunas de las funciones de entrada salida provistas por el lenguaje.

11. Archivos

Un archivo es una colección de datos relacionados. C considera a un archivo como una serie de bytes y ofrece, en su librería, un gran número de rutinas o funciones que permiten el manejo de los mismos. El código de esas rutinas se encuentra declarado en el archivo estándar < stdio.h> por lo tanto antes de hacer cualquier tarea que involucre el uso de archivos es necesario incluir al comienzo del programa la línea:

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

Por razones técnicas, se debe definir un bloque de memoria o estructura en el que se almacenará toda la información asociada con el archivo. Esta estructura ya está definida en el archivo de encabezamiento `stdio.h` y recibe el nombre de `FILE`. Siempre se usará con punteros a la misma.

Por lo tanto, solo es necesario declarar un puntero a esta estructura llamado *puntero de manejo de archivo*. A través del mismo se podrá acceder a la información del archivo. La única declaración necesaria para un puntero FILE se ejemplifica por:

```
FILE *fp;
```

Esta instrucción permite declarar a *fp* como un puntero a un FILE.
Por ejemplo:

```
#include <stdio.h>
FILE *entrada;
```

C provee 3 archivos estándar: la entrada, la salida y el error, cuyos punteros de manejo son:

Archivos	Descripción
stdin	entrada estandard (abierto para lectura)
stdout	salida estandar (abierto para escritura)
stderr	archivo de error estandard (abierto para escritura)

La pregunta es cómo hacer para trabajar con los archivos del usuario, esto es, cómo conectar las funciones que manejan a los datos con los nombres externos que un usuario tiene en mente.

Las reglas son simples. Para trabajar con un archivo, primero se debe conectar el programa con el archivo, a través de una operación de apertura (open), la cual toma un nombre externo como datos.dat o texto.txt, hace algunos arreglos y negociaciones con el sistema operativo (cuyos detalles no deben importar), y regresa un puntero que será usado en posteriores lecturas o escrituras del archivo. Al finalizar, es necesario desconectarlo a través de la operación de cerrado (close).

Función fopen

Sintaxis:

```
variable_file = fopen(nombre,modo)
```

donde:

variable_file: es una variable del tipo FILE.
nombre: es el nombre actual del archivo (data.txt, temp.dat)
modo: señala cómo se pretende emplear el archivo. Los modos disponibles son:

- r: sólo lectura. El archivo debe existir.
- w: se abre para escritura, se crea un archivo nuevo, o se sobrescribe si ya existe.
- a: añadir, se abre para escritura, el cursor se sitúa al final del archivo. Si el archivo no existe, se crea.
- r+: lectura y escritura. El archivo debe existir.
- w+: lectura y escritura, se crea un archivo nuevo, o se sobrescribe si existe.
- a+: añadir, lectura y escritura, el cursor se sitúa al final del archivo. Si el archivo no existe, se crea.

La función fopen retorna un puntero de manejo del archivo el cual va a ser usado en las subsiguientes operaciones de I/O. En caso de producirse un error de I/O, retorna el valor null. La detección del error se puede hacer como sigue:

```
FILE *in_file; /* archivo para la lectura */
in_file = fopen("input.txt", "r");
if ( in_file == NULL ){ /* test por error */
```

```
        fprintf(stderr,"Error al abrir el archivo input.txt");
        exit(1); /* ver Apendice D */
    }
```

Función fclose

Sintaxis:

```
int fclose(FILE *archivo)
```

La función fclose es el inverso de fopen, interrumpe la conexión establecida por fopen entre el manejador de archivo y el nombre del archivo, liberando al puntero manejador para ser usado con otro archivo.

Como algunos sistemas operativos limitan la cantidad de archivos que pueden estar abiertos al mismo tiempo, es aconsejable liberar los punteros manejadores cuando ya no son necesarios. Por otro lado, cuando se ejecuta el fclose se almacenan los datos que se encuentran en el buffer de memoria y actualiza algunos datos del archivo que son manejados por el sistema operativo.

Un valor de retorno 0 significa que no se produjo ningún error y ha sido cerrado correctamente.

Función feof

Sintaxis:

```
int feof(FILE *archivo)
```

Esta función permite verificar si se ha llegado al fin de `archivo`. Frecuentemente es necesario trabajar con datos los cuales se encuentran almacenados en forma secuencial. Por lo tanto la manera de accederlos es permanecer leyendo mientras no se detecte el fin de archivo. La función feof(fp) retorna un valor distinto de 0 si se alcanza el fin de archivo, o 0 en otro caso.

Por ejemplo:

```
while (!feof(fdb))
```

Mientras no se alcance el fin de archivo, la función feof(fdb) retornará el valor 0, el cual al ser negado con el operador **!**, se vuelve 1 convirtiendo la condición de la sentencia **while** en verdadero.

11.1. Archivos ASCII y Binarios

Los archivos ASCII consisten en texto legible (archivos de texto). Cuando se escribe un programa, por ejemplo prog.c, se está creando un archivo del tipo ASCII.

Las computadoras trabajan sobre datos binarios. Cuando se desea leer un caracter de un archivo ASCII, el programa debe procesar el dato a través de rutinas de conversión que implican un costo muy alto. Los archivos binarios no requieren de ninguna conversión y generalmente ocupan menos espacio que un archivo ASCII.

Los archivos ASCII generalmente son portables, ellos pueden ser movidos de una máquina a otra con mínimos problemas, mientras que en el caso de los archivos binarios es más complicado.

Qué tipo de archivos usar? En la mayoría de los casos es preferible usar archivos ASCII, permiten chequear fácilmente si el dato es correcto y si es un archivo cuyo tamaño es relativamente chico y el tiempo de conversión no es significativo en la performance total del programa.

En caso de trabajar con grandes cantidades de datos es conveniente usar el formato binario.

Los archivos ASCII pueden ser leídos, creados o modificados directamente con un editor de textos. Las funciones que provee el lenguaje C para su manejo son simples de usar. La desventaja es que los archivos son usualmente más grandes. Por ejemplo, en una computadora en la que el tamaño de una palabra es 16 bits, el entero 25000 en representación ASCII ocupa 5 bytes, más los espacios que se necesitan para separarlo de los otros datos. En su representación binaria ocupa sólo dos bytes y no necesita separadores.

11.2. Trabajo con los archivos

Hasta ahora los programas desarrollados reciben datos de la entrada estándar y devuelve resultados escritos en la salida estándar, los cuales son definidos automáticamente para los programas por el sitema operativo local.

El siguiente paso consiste en trabajar con un archivo que ya está conectado al programa.

11.2.1. Archivos ASCII

Función fprintf (Salida con formato)

Sintaxis:

```
int fprintf(FILE *archivo, const char *formato,...);
```

Es equivalente a la función printf, pero la salida se dirige a un archivo en lugar de la salida estándar.

Esta función toma una salida, la convierte y la escribe en el archivo bajo el control de formato. El valor *int* que devuelve la función representa el número de caracteres escritos o un valor negativo en caso de que ocurra algún error. La cadena de *formato* contiene 2 tipos de elementos:

1. Caracteres ordinarios que son copiados textualmente en la salida, y
2. Las especificaciones de conversión, las cuales provoca la conversión e impresión de los sucesivos argumentos de la función fprintf. Cada especificación de conversión comienza con el caracter % seguido del caracter de conversión. (Ver apéndice B)

Por ejemplo:

```
FILE *fp;
fp = fopen (‘‘datos.txt’’, ‘‘w+’’);
char letra;
letra = ’M’;
fprintf(fp, ‘‘El codigo ascii de %c es %d\n’’, letra, letra);
```

Función fscanf (Entrada con formato)

Sintaxis:

```
int fscanf (FILE *archivo, const char* formato, ....);
```

Equivalente a scanf, pero la entrada se toma de un archivo en lugar del teclado (o entrada estándar stdin).

Esta función lee de *archivo* bajo el control de formato, y asigna los valores convertidos a los argumentos subsecuentes, cada uno de los cuales debe ser un puntero.

Cuando *formato* se ha agotado la función termina de ejecutarse.

La función puede devolver como resultado:

- EOF si encuentra el fin del archivo o un error antes de la conversión
- Un valor *int* que representa el número de items de entrada convertidos y asignados.

La cadena de formato contiene especificaciones de conversión a aplicar al campo de entrada. Consiste en el símbolo % asociado con un caracter que indica el formato deseado (ver apéndice B).
Por ejemplo

```
FILE *fp;
fp = fopen (‘‘datos.txt’’, ‘‘r’’);
int dia, anio;
char mes[20];
fscanf(fp, ‘‘%d %s %d’’, &dia, mes, &anio);
```

11.2.2. Archivos Binarios

Función fwrite (Salida)

Sintaxis

```
size_t fwrite(void *puntero, size_t tam, size_t nreg, FILE *archivo)
```

Permite escribir en *archivo* uno o varios objetos(*nreg*) del mismo tamaño (*tam*) apuntado por *puntero* que señala a la zona de memoria donde se encuentran los datos leídos.

La función retorna el número de objetos escritos (de tipo `size_t`), no el número de bytes. En caso de error el número que devuelve es menos que *nreg*

Función fread (Entrada)

Sintaxis

```
size_t fread(void *puntero, size_t tam, size_t nreg, FILE *archivo)
```

Permite leer de un *archivo* uno o varios objetos (*nreg*) de igual tamaño (*tam*) y almacenarlos en la posición de memoria señalada por *puntero*.

El usuario debe asegurarse de tener suficiente espacio para contener la información leída.

La función retorna el número de registros leídos.

Existen además otras funciones:

- Para la detección de errores:

  `ferror(fp)` retorna un valor distinto de 0 si ha ocurrido un error durante la lectura o escritura en archivo, o 0 en otro caso.

- Funciones que permiten trabajar caracter a caracter:

  `getc(fp)` retorna el próximo caracter en el archivo, o `EOF` si no hay más.

  `putc(c,fp)` escribe el caracter `c` en el archivo.

- Funciones que trabajan con la posición física en el archivo:

  `fseek(fp,desplazamiento,origen)` permite modificar la posición de lectura y escritura en el archivo. Sus argumentos son el puntero al archivo *fp*, un desplazamiento en bytes *desplazamiento* y un valor *origen* que especifica el origen del desplazamiento. La función retorna 0 si la operación es exitosa y un valor distinto de 0 en otro caso.

  Los valores posibles para *origen* son:

origen	descripción
`SEEK_SET`	el desplazamiento es considerado desde el comienzo.
`SEEK_CUR`	el desplazamiento es considerado desde la posición corriente.
`SEEK_END`	el desplazamiento es considerado desde el final.

  `ftell(fp)` retorna la posición corriente en el archivo apuntado por *fp*.

11.3. Ejemplos

El siguiente programa lee enteros del archivo `num.dat` y escribe sus valores absolutos en el archivo `abs.dat`:

```
#include <stdio.h>

int main() {
  FILE *in,*out;
  int i;

  if ((in = fopen("num.dat","r")) == NULL) {
    printf("Error: el archivo num.dat no se puede abrir\n");
    exit(1);
  }

  if ((out = fopen("abs.dat","w")) == NULL) {
    printf("Error: el archivo abs.dat no se puede abrir\n");
    exit(1);
  }

  while (fscanf(in,"%d",&i) > 0)
    fprintf(out,"%d",abs(i));

  fclose(in);
  fclose(out);
  return(0);
}
```

Las dos últimas sentencias corresponden a la invocación de la función `fclose`, la cual cierra la conexión entre el programa y el archivo cuyo puntero es pasado como argumento.

Con estas funciones, la información siempre se escribe en formato ASCII en el archivo.

El siguiente programa permite copiar un archivo en otro. Asume que los nombres de los archivos han sido pasados como argumentos en la línea de comandos:

```
#include <stdio.h>

int main(int argc,char *argv[]) {
  FILE *in,*out;
  int c;

  if ((in = fopen(argv[1],"r")) == NULL) {
    printf("Error: el archivo de entrada no se puede abrir\n");
    exit(1);
  }

  if ((out = fopen(argv[2],"w")) == NULL) {
    printf("Error: el archivo de salida no se puede abrir\n");
    exit(1);
  }

  while (c = getc(in),c != EOF)
    putc(c,out);
```

```
  fclose(in);
  fclose(out);
  return(0);
}
```

El siguiente programa muestra como es posible grabar un arreglo de registros en forma completa:

```
#include <stdio.h>

struct punto {
  int x;
  int y;
} p[2] = { { 2 , 3 } , { 1 , 8 } };

int main() {
  FILE *fp;

  if ((fp = fopen("arch.dat","w")) == NULL) {
    printf("Error: el archivo arch.dat no se puede abrir\n");
    exit(1);
  }

  if (fwrite(p,sizeof(struct punto),2,fp) != 2) {
    printf("Error: no se pudieron grabar las dos componentes\n");
    exit(1);
  }

  fclose(fp);
  return(0);
}
```

El siguiente programa muestra como se pueden leer los registros almacenados en el archivo generado con el programa anterior:

```
#include <stdio.h>

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

int main() {
  FILE *fp;

  if ((fp = fopen("arch.dat","r")) == NULL) {
    printf("Error: el archivo arch.dat no se puede abrir\n");
    exit(1);
  }

  while (fread(&p,sizeof(struct punto),1,fp) == 1) {
    printf("x = %d y = %d\n",p.x,p.y);
  }
```

```
  fclose(fp);
return(0);
}
```

Notar que aunque las estructuras fueron almacenadas juntas, se pueden leer una por una. Notar además que el valor retornado por la función `fread` se utiliza para detectar el fin de archivo. Si la función `fread` retorna un valor distinto al número de componentes que se solicitó que leyera (1 en este caso), implica que no pudo realizar una lectura satisfactoria.

El siguiente programa muestra como se puede escribir al final del archivo que fue generado anteriormente, y luego leer todas las componentes:

```
#include <stdio.h>

struct punto {
  int x;
  int y;
} p,q = { 4 , 5 };

int main() {
  FILE *fp;

  if ((fp = fopen("arch.dat","r+")) == NULL) {
    printf("Error: el archivo arch.dat no se puede abrir\n");
    exit(1);
  }

  if (fseek(fp,0,SEEK_END) != 0) {
    printf("Error: no se pudo ir al final del archivo\n");
    exit(1);
  }

  if (fwrite(&q,sizeof(struct punto),1,fp) != 1) {
    printf("Error: no se pudo grabar la estructura\n");
    exit(1);
  }

  if (fseek(fp,0,SEEK_SET) != 0) {
    printf("Error: no se pudo ir al comienzo del archivo\n");
    exit(1);
  }

  while (fread(&p,sizeof(struct punto),1,fp) == 1) {
    printf("x = %d y = %d\n",p.x,p.y);
  }

  fclose(fp);
return(0);
}
```

12. Administración dinámica de memoria

La administración dinámica de memoria es el proceso por el cual la memoria puede ser solicitada y liberada en cualquier punto durante la ejecución del programa. Para poder utilizar las funciones que provee C se debe incluir la biblioteca `stdlib.h`. Entre las funciones más importantes están:

```
void *malloc(size_t n);
void free(void *p);
```

La función `malloc` se utiliza para pedir un bloque de memoria de tamaño `n` (en bytes). Si existen `n` bytes consecutivos disponibles, retorna un puntero al primer byte. Si no hubiera disponibilidad, retorna el puntero nulo `NULL`.

Por ejemplo, si queremos copiar el string `b` en `a`, que es sólo un puntero y no tiene área asignada para copiar los caracteres, se puede hacer:

```
char *a;
char b[] = "hola";

if ((a = (char*)malloc(strlen(b)+1)) == NULL) {
  printf("no hay memoria suficiente\n");
  exit(1);
}

strcpy(a,b);
```

La función `malloc` se invoca con la longitud del string `b` mas uno. Esto se debe a que se requiere un espacio capaz de contener todos los caracteres del string `b` mas el caracter nulo. La función `malloc` retorna un puntero a `void` que apunta a ese área. Este resultado debe entonces ser convertido a un puntero a `char`, lo que se logra con el cast ubicado delante de `malloc`. El resultado de esta conversión es entonces asignado al puntero `a`. El valor es inmediatamente comparado con `NULL`, dado que si es igual, indica que no hay memoria suficiente para cumplir con el requerimiento. Sólo cuando se está seguro que `a` apunta a un área adecuada, la copia puede tomar lugar.

La función `exit` que está usada luego de imprimir el mensaje de error, sirve para terminar la ejecución de un programa. El valor que se pasa como argumento es el valor de retorno de la función `main` al sistema operativo (recordar lo comentado en la sección 2).

Suponga que se necesita obtener espacio para `n` enteros durante la ejecución del programa. La función `malloc` requiere el tamaño del área expresada en bytes. Para conocer cuantos bytes utiliza un entero se puede usar el operador `sizeof`, que toma el nombre de un tipo o un objeto de datos como argumento, y retorna su tamaño en bytes.

Por ejemplo, el siguiente trozo de programa asigna dinámicamente espacio para un arreglo de `n` enteros, e inicializa todas sus componentes a 0:

```
int *arr,n,i;

scanf("%d",&n);

if ((arr = (int*)malloc(sizeof(int) * n)) == NULL) {
  printf("no hay memoria suficiente\n");
  exit(1);
}
for(i = 0;i < n;i++) arr[i] = 0;
```

El argumento de la función `malloc` es el tamaño en bytes de un entero multiplicado por el número de enteros. Notar que ahora el puntero retornado por `malloc` debe ser transformado a un puntero a enteros.

En el ejemplo siguiente, se reserva lugar para `n` estructuras de tipo `empleado` (como fuera definido en la sección 6.2):

```
int n;
struct empleado *p;

scanf("%d",&n);

if ((p = (struct empleado*)malloc(sizeof(struct empleado) * n))
          == NULL) {
  printf("no hay memoria suficiente\n");
  exit(1);
}
for(i = 0;i < n;i++) {
  p[i].codigo = 0;
  p[i].sueldo = 0.0;
}
```

La función de biblioteca `free` se utiliza para retornar la memoria que fue obtenida a través de `malloc`. Por ejemplo, para retornar toda la memoria que fue asignada en los ejemplos previos, se puede ejecutar:

```
free((void*)a);
free((void*)arr);
free((void*)p);
```

El puntero debe ser transformado a través de un cast a puntero a `void` dado que es el tipo de argumento que espera la función `free`.

Pregunta 36 Explique lo que hace el siguiente programa y diga cual es la salida impresa:

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

int main() {
  int i;
  char *s;
  int j;

  for(i = 2;i < 6;i++) {
    if ((s = (char*)malloc(i+1)) == NULL) {
      printf("no hay memoria suficiente\n");
      exit(1);
    }
    for(j = 0;j < i;j++) s[j] = 'a';
    s[j] = '\0';
    printf("%s\n",s);
    free((void*)s);
  }
  return(0);
}
```

13. Parámetros del programa

Es posible acceder desde el programa a los argumentos que se pasan en la línea de comandos cuando se invoca el programa para su ejecución. Por ejemplo, si el programa se llama `programa`, puede ser invocado, por

ejemplo, desde la línea de comandos:

```
$ programa a.txt 5 hola
```

Los argumentos que han sido pasados al programa pueden ser accedidos a través de dos argumentos opcionales de la función `main`:

```
int main(int argc,char *argv[]) ...
```

El primero, llamado por convención `argc`, es el que indica el número de argumentos en la línea de comandos, y el segundo, llamado `argv`, es el vector de argumentos. `argc` es un entero y `argv` es un arreglo de strings (o equivalentemente un arreglo de punteros a caracteres). Los valores de `argc` y `argv` para el ejemplo anterior se muestran en la figura 16.

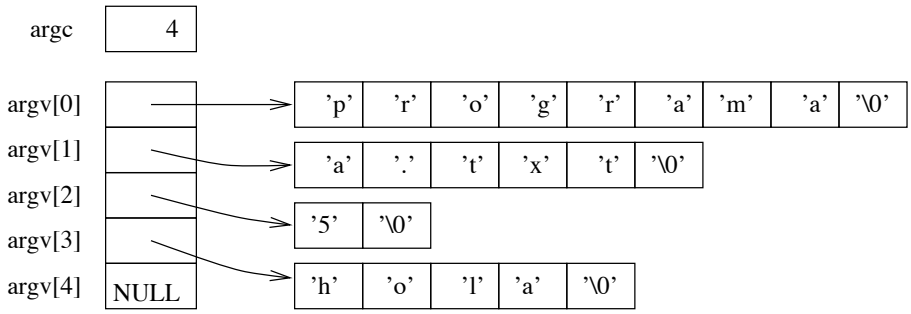

Figura 16: parámetros del programa

El siguiente programa imprime todos los strings que han sido pasados como argumento:

```
#include <stdio.h>

int main(int argc,char *argv[]) {
  int i;

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

Si al compilarlo, se le da el nombre `programa` a la versión ejecutable, se obtiene como resultado de la invocación mostrada al comienzo:

```
programa
a.txt
5
hola
```

Notar que el nombre del programa es el primer string en el arreglo de valores `argv`. También el ANSI estándar garantiza que `argv[argc]` es `NULL`.

Notar también que la expresión `char *argv[]` es totalmente equivalente a `char **argv`.

14. Declaraciones de tipos

La declaración `typedef` permite darle un identificador o nombre a los tipos, de forma tal que puedan ser utilizados más adelante. Por ejemplo:

```
typedef int integer;
```

define el tipo `integer` [5] como el tipo estándar `int`, asi que ahora se puede definir una variable de tipo entero usando `integer`:

```
integer i;
```

Un ejemplo un poco más útil es el siguiente:

```
typedef int array[100];
```

aqui `array` es el tipo "arreglo de 100 enteros"[6], entonces al declarar posteriormente:

```
array a;
```

`a` es un arreglo de 100 enteros.
Por ejemplo:

```
typedef struct {
  float x;
  float y;
} punto;
```

declara el tipo `punto`[7], que puede ser utilizado para declarar una variable de tipo arreglo de 10 puntos:

```
punto b[10];
```

sin usar la palabra `struct` como antes (recordar la sección 6.2).

15. Estructura del programa

Las variables tienen dos atributos: la clase de almacenamiento y su alcance, los cuales son detallados a continuación.

15.1. Clases de almacenamiento

Ya se vió que un programa consiste de un conjunto de funciones. Todas las variables que se han utilizado hasta el momento eran locales a esas funciones. Cuando el programa no está ejecutando una función, sus variables locales ni siquiera existen. El espacio se crea para ellas en forma automática cuando la función se invoca. El espacio se destruye también en forma automática cuando la ejecución de la función termina. Se dice que estas variables tiene una clase de almacenamiento *automática*.

Se pueden definir también variables locales a funciones que sean capaces de retener el valor entre las sucesivas invocaciones de la función en la que han sido definidas, aun cuando no puedan ser accedidas fuera de ella. Se dice que estas variables tienen una clase de almacenamiento *estática*.

Por ejemplo, en el siguiente programa la variable `b` tiene clase de almacenamiento estática y la variable `a` tiene clase automática (o sea es una variable local normal):

[5] Una regla que permite entender esta sintaxis un poco exótica de C es la siguiente: tape con el dedo la palabra `typedef`. Lo que queda es la declaración de una variable entera llamada `integer`. Al colocar la palabra `typedef` nuevamente, en lugar de variable, la palabra `integer` es un tipo.

[6] Tape nuevamente la palabra `typedef`, lo que queda es la declaración de la variable `array`, de tipo arreglo de 100 enteros. Al colocar nuevamente la palabra, no es una variable, sino un tipo.

[7] ¡Use la regla!

```
#include <stdio.h>

void f() {
  int a = 0;
  static int b = 0;

  printf("a = %d b = %d\n",a++,b++);
  return;
}

int main() {
  f();
  f();
  return(0);
}
```

El programa tiene una función `f` que no recibe argumentos y no retorna ningún valor. Define dos variables locales `a` y `b` ambas inicializadas a 0. `a` tiene clase automática, por lo que es creada e inicializada cada vez que se ejecuta la función `f`. `b` tiene clase estática y por lo tanto será capaz de retener su valor entre las invocaciones de `f`, y sólo será inicializada la primera vez. Los valores impresos por el programa son:

```
a = 0 b = 0
a = 0 b = 1
```

15.2. Alcance

Es posible definir variables que se pueden acceder sólo dentro del bloque definido por una sentencia compuesta, por ejemplo:

```
void f() {
  int i;

  for(i = 0;i < 10;i++) {
    char c,i;

    i = 'a';
    c = 'b';
  }
  return;
}
```

La variable `c` se crea al iniciar cada iteración y se destruye al final de ella. La variable `i` definida como `char` dentro de la iteración también es local a ella, e inhibe el acceso a la variable `i` definida local a la función. Este alcance se denomina alcance de *bloque*.

Es posible definir variables que se pueden acceder en más de una función, por ejemplo, la variable `a` del programa puede ser accedida en todas las funciones:

```
#include <stdio.h>

static int a;

void f() {
  printf("%d\n",a);
```

```
  return;
}

void g() {
  char a;
  a = 'a';
  return;
}

int main() {
  a = 12;
  f();
  return(0);
}
```

En la función `g`, sin embargo, al estar redefinida, toma precedencia la definición local, y todas las referencias a la variable `a` dentro de `g` corresponden a la variable `a` local. Esta regla de alcance se denomima alcance de *archivo*, dado que todas las funciones del archivo que componen el programa pueden acceder a la variable.

Los programas en C pueden estar compuestos por varios archivos fuente. Pueden ser compilados en forma separada, y los archivos objeto (ya compilados) obtenidos son unidos a través de un linker en un único programa ejecutable. Se permite que las variables y funciones declaradas en un archivo, se accedan desde otros. Este alcance se denomina alcance de *programa*, dado que pueden ser accedidas desde cualquier parte del programa. Estos objetos se denominan externos, porque son definidos fuera del módulo en el que se utilizan.

Considere el siguiente programa, compuesto por dos archivos: `uno.c` y `dos.c`:

```
/* archivo uno.c */
int a;
static float b;

int main() {
  int f(int,float);
  extern float g(int);

  b = 3.9 + g(2);
  a = f(2,b);
  return(0);
}

int f(int x,float y) {
  return a + b + x;
}
```

```
/* archivo dos.c */
extern int a;
extern int f(int,float);

float g(int x) {
  return (x + a + f(x,3.1)) / 2.0;
}
```

En el archivo `uno.c` se declaran dos variables fuera del alcance de las funciones. La variable `b` es estática. Esto significa que puede ser accedida en todas las funciones, pero sólo dentro del archivo en el que ha sido definida.

La variable `a` en cambio, es una variable externa, y puede ser accedida en el archivo en el que está definida, y en todos los archivos en los que esté declarada. Su definición aparece en el archivo `uno.c` y su declaración en el archivo `dos.c`.

Notar la diferencia entre definición y declaración. Una declaración especifica los atributos solamente, mientras que una definición hace la misma cosa, pero también asigna memoria. Notar que puede haber entonces sólo una definición y múltiples declaraciones, cada una en uno de los archivos desde los que se desea acceder a la variable.

La declaración de `a` que aparece en el archivo `dos.c` permite que la variable `a` pueda ser accedida en todas las funciones de ese archivo.

El mismo concepto se aplica a las funciones. Para poder acceder a la función `g` en el archivo `uno.c`, se realiza una declaración en el archivo `uno.c`. Notar que la declaración está dentro de la función `main`. Esto quiere decir que la función `g` sólo se puede invocar desde `main` y no desde `f`.

Puede llamar la atención la ocurrencia de la declaración de `f` dentro de `main` si ambas están declaradas en el mismo archivo. La razón de esta declaración, que se llama *prototipo*, es que el compilador procesa el texto desde arriba hacia abajo, y si encuentra primero la invocación a `f` antes de su definición no puede determinar si el uso es correcto o no. Una solución es usar prototipos, la otra es colocar la definición de la función `f` primero.

16. El preprocesador

Es un programa independiente que realiza algunas modificaciones al texto del programa antes que éste sea analizado por el compilador. Los comandos para el preprocesador comienzan con un símbolo `#` en la primera columna del texto. El preprocesador modifica el texto de acuerdo a lo especificado por los comandos y los elimina. Es decir, que el compilador nunca recibe un comando que empieze con `#`.

Por ejemplo, la directiva `#include` usada en los programas, le indica al preprocesador que debe incorporar en ese punto del texto, el contenido del archivo que se especifica a continuación. Por ejemplo, el uso:

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

causa que todas las declaraciones y prototipos de las funciones de la biblioteca de entrada/salida estándar se incorporen en ese punto, por lo que podrán ser referenciadas posteriormente por nuestro código.

La directiva `#define` permite definir constantes. Por ejemplo:

```
#define N 10

int a[N];
int b[N];

int main() {
  int i;

  for(i = 0;i < N;i++) a[i] = b[i] = 0;
  return(0);
}
```

Este programa es tomado por el preprocesador y transformado en:

```
int a[10];
int b[10];

int main() {
  int i;

  for(i = 0;i < 10;i++) a[i] = b[i] = 0;
  return(0);
}
```

y recién entonces es compilado. Notar que el compilador no sabe que existe una constante que se llama `N`.

La sustitucion es textual, es decir, de un texto por otro, en este caso de `N` por 10. Es importante tener en cuenta este punto, dado que si por ejemplo, se hubiera puesto inadvertidamente un punto y coma al final de la declaración de la constante, se produciría un error difícil de determinar:

```
#define N 10;

int a[N];
...
```

sería reemplazado por:

```
int a[10;];
```

y como el error es marcado en el texto original, aparecería:

```
#define N 10;

int a[N];
       ^ falta un corchete (se leyo punto y coma)
```

En realidad, la directiva `#define` es más poderosa y permite definir macros con argumentos, haciendo también una sustitución textual de ellos. Por ejemplo:

```
#define cuadrado(x)  x * x
...
i = cuadrado(2);
```

El código se reemplaza por:

```
i = 2 * 2;
```

hay que tener cuidado, ya que no siempre la salida es la esperada. Por ejemplo:

```
i = cuadrado(2 + a);
```

La sustitución será:

```
i = 2 + a * 2 + a;
```

que no es lo esperado. La solución consiste en usar paréntesis en la declaración de la macro cada vez que aparece el argumento, y luego abarcando toda la expresión. Por ejemplo:

```
#define cuadrado(x) ((x)*(x))
...
i = cuadrado(2 + a);
```

se sustituye por:

```
i = ((2 + a) * (2 + a));
```

Pregunta 37 El siguiente programa da un mensaje de advertencia al ser compilado: posible uso de la variable `cont` antes de asignarle un valor. ¿Cuál es la razón?.

```
#include <stdio.h>
#define MAX =10

int main() {
  int cont;

  for(cont =MAX;cont > 0;cont--) printf("%d\n",cont);
  return(0);
}
```

Pregunta 38 ¿Cuántas veces es incrementado el contador en cada iteración?

```
#include <stdio.h>
#define cuadrado(x) ((x)*(x))

int main() {
  int cont = 0;

  while(cont < 10) {
    printf("%d\n",cuadrado(cont++));
  }
  return(0);
}
```

A. Precedencia y asociatividad de las operaciones

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

Los operadores que están en la misma línea tienen la misma precedencia. La precedencia disminuye de arriba hacia abajo, eso quiere decir, que por ejemplo:

```
a + b * c
```

se ejecuta como `a + (b * c)` y no como `(a + b) * c`, dado que el operador de producto tiene mayor precedencia que el de la suma.

La asociatividad tiene que ver con el orden en que se evalúan los argumentos para operadores de la misma precedencia. Por ejemplo:

```
a = b = c;
```

se ejecuta como `a = (b = c)`, dado que la asociatividad es de derecha a izquierda.

B. Especificadores de formato

Estos son los especificadores de formatos más comunes.

CARACTER	SIGNIFICADO
`%d`	entero decimal
`%x`	entero hexadecimal
`%u`	unsigned
`%c`	caracter
`%s`	string
`%f`	flotante
`%%`	un signo %

C. Respuestas

Respuesta 1 El programa imprime:

```
Esta es
una prueba
```

Notar que el caracter `\n` puede estar en cualquier lugar del string y que los espacios son significativos.

Respuesta 2 El error se produce ya que no se ha incluído la instrucción

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

que contiene la definicion de la función `printf`. Ver página 4.

Respuesta 3 El programa imprime:

```
255 ff
```

Respuesta 4 Si, dado que sólo se pide que el rango de las variables enteras `short` no sea mayor que el de un entero convencional, y el de las variables enteras `long` no sea menor al de un entero convencional.

Respuesta 5 El programa imprime:

```
A B C
```

La variable `c` es inicializada con el caracter `'A'`, o con el entero 65 que es su código ASCII. La expresión `c+1` tiene valor 66, y corresponde al caracter sucesivo, o sea `'B'`. De la misma forma, `c+2` representa el caracter `'C'`.

Respuesta 6 A la variable `a` se le asigna el valor 58 decimal, obtenido sumando el hexadecimal `0x28` (decimal 40), el octal `010` (decimal 8) y el decimal `10`. A la variable `b` se le asigna el decimal 320, obtenido sumando el hexadecimal `0xff` (decimal 255) y el código ASCII del caracter `'A'` (decimal 65).

Respuesta 7 La constante `08` comienza con 0 por lo que es una constante octal. Sin embargo, el dígito 8 no puede ser usado en las constantes octales.

Respuesta 8 Sólo la segunda. El rango para las constantes enteras en una implementación de 16 bits es de -32768...32767, por lo que la constante entera 50000 está fuera de rango. Sin embargo, el rango para las `unsigned` es de 0...65535.

Respuesta 9 Los resultados son los siguientes:

expresión	tipo	resultado	comentario
`10 + 'A' + 5u`	`unsigned`	80u	
`50000u + 1`	`unsigned`	50001u	
`50000 + 1`		error	fuera de rango `int`

Respuesta 10 Se presentan problemas cuando el valor no puede representarse adecuadamente en el nuevo tipo. Por ejemplo:

```
float f = 3.5;
int i;

i = f;
```

Si bien la operación se permite, el valor asignado es 3 y no 3.5.

Respuesta 11 No, no hay forma de evitar el uso del cast. Dado que `i` y `j` son variables enteras, la expresión `i / j` siempre será evaluada como división entera. De hecho, este ejemplo muestra más la utilidad del cast que el ejemplo de la sección 3.8.

Respuesta 12 A la variable `b` se le asigna el valor 5 (entero)y a la variable `g` se le asigna el valor 1.5. Notar que si bien podría haber pérdidad de precisión al convertir `f` a entero, el programador demuestra que es consciente de ello al utilizar el cast.

Respuesta 13 Las expresiones tienen los valores 1, 1 y 8 respectivamente.

Respuesta 14 Las expresiones tienen los valores 1, 1, 1 y 0 respectivamente.

Respuesta 15 La salida es:

```
45 | 71 = 75
```

Respuesta 16 En caso que el tipo entero tenga 16 bits, 0xffff tiene 16 bits en 1 y por lo tanto cumple con el objetivo. Sin embargo, si el entero tuviera 32 bits, por ejemplo, sólo los 16 bits menos significativos de `a` tendrán el valor 1.

La segunda asignación, en cambio, es portable, dado que `~0` indica que todos los bits del número 0, que son ceros, deberán ser complementados obteniéndose unos, independientemente de cuantos sean.

Respuesta 17 Las expresiones en forma completa son:

versión compacta	normal
`a |= 0x20;`	`a = a | 0x20`
`a <<= (b = 2);`	`a = a << (b = 2);`

Respuesta 18 Los valores finales de las variables son:

a 3
b 3
c 17
d 10
i 9
j 6

y las expresiones en forma completa son:

versión compacta	normal
`a = i++ - j++`	`a = i - j`
	`i = i + 1;`
	`j = j + 1;`
`b = ++i - ++j`	`i = i + 1;`
	`j = j + 1;`
	`b = i - j;`
`c = (d = i--) + j--;`	`d = i;`
	`i = i - 1;`
	`c = d + j;`
	`j = j - 1;`

Respuesta 19 Los valores finales de las variables son:

a	3
b	2
c	4
d	mantiene el valor anterior
i	2
j	4

Respuesta 20 Corresponde al segundo `if`. La forma en la que está indentado el fragmento de programa no es la correcta. Debería haber sido:

```
if (i < 5)
  if (j < 8)
    printf("uno\n");
  else
    printf("dos\n");
```

Para evitar la posibilidad de confusión es conveniente usar llaves auxiliares aunque no sean necesarias. Por ejemplo:

```
if (i < 5) {
  if (j < 8)
    printf("uno\n");
  else
    printf("dos\n");
}
```

Respuesta 21 Este programa ilustra uno de los errores más comunes y frustrantes. El problema es que la sentencia:

```
if (balance = 0)
```

contiene una asignación en lugar de una comparación. Debería haber sido escrita como:

```
if (balance == 0)
```

Respuesta 22 El programa imprime 9 cuando `i` vale inicialmente 1, y 4 cuando `i` vale inicialmente 3.

Respuesta 23 Para los valores 0 y 1 es trivial dado que no ingresa a la iteración. Para el valor 4 el `for` se ejecuta tres veces.

Respuesta 24 Reescrito usando for:

```
/* cuenta el numero de espacios */
#include <stdio.h>

int main() {
  int c,nro = 0;

  for(c = getchar();c != EOF,c = getchar()) {
    if (c == ' ') nro++;            /* es un espacio */
  }
  printf("numero de espacios = %d\n",nro);
  return(0);
}
```

Respuesta 25 El problema es que se ha colocado un punto y coma al final del paréntesis del `for`. C interpreta entonces que el `for` tiene un cuerpo vacío, y ejecuta la iteración sin imprimir hasta que el valor de `c` supera el límite, o sea Z. El próximo valor es entonces impreso.

Respuesta 26 El programa cuenta el número de caracteres distintos de newline y espacios (en la variable `chars`), el número de letras mayúsculas (en `may`) y el de letras minúsculas (en `min`). Se detiene cuando se lee el caracter `EOF`. Notar que el 1 en la condición del `while` tiene el efecto de hacer que siempre la condición se evalúe como verdadera. No es un loop infinito dado que hay forma de salir a través del `break`.

Respuesta 27 Se podría escribir como sigue:

```
#include <stdio.h>

int main() {
  int i,c;
  int min = 0,may = 0,chars = 0;

  c = getchar();
  while (c != EOF) {

    if (c != ' ' && c != '\n') {
      if (c >= 'A' && c <= 'Z') may++;
      if (c >= 'a' && c <= 'z') min++;
      chars++;
    }
    c = getchar();
  }
  return(0);
}
```

Respuesta 28 La descripción de lo que ocurre en cada asignación se presenta a continuación:

asignación	comentario
`val.valor_float = 2.1;`	legal
`val.valor_int = 4;`	legal
	se sobreescribe el valor anterior
`i = val.valor_int;`	legal
`f = val.valor_float;`	no legal
	resultado inesperado (basura)
`val.valor_float = 3.3;`	legal
	borra el valor del campo entero
`i = val.valor_int;`	no legal
	resultado inesperado (basura)

Notar que todas las operaciones son válidas y permitidas por el compilador. El uso de la palabra legal o no legal tiene que ver con la semántica que se busca a las operaciones. Desde el punto de vista sintáctico el programa es perfecto y todas las asignaciones son de enteros a enteros y de flotantes a flotantes.

Respuesta 29 Se accede:

```
prod.productos[3] = 23;
```

Respuesta 30 No existe ninguna restricción, dado que el hecho de que el código del país se utilize solamente para productos importados es algo que está en la mente del programador y no en la descripción sintáctica

del programa. El programa no fuerza este uso, sino que debería ser tenido siempre en cuenta por el programador. Si se utiliza el campo `cod_pais`, se destruirá el valor almacenado en el campo `cod_prov` y viceversa.

Respuesta 31 Imprime:

```
6 6 6
7 7 7
```

Respuesta 32 La representación gráfica se presenta en la figura 17.

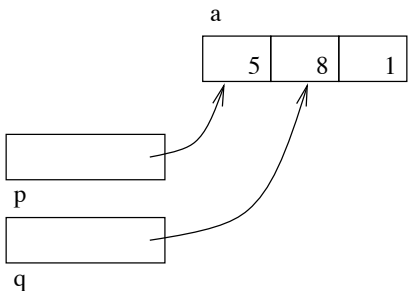

Figura 17: arreglo y punteros

Los resultados de las expresiones se presentan a continuación:

asignación	comentario
`i = *p + *q;`	asigna el valor 13
`*p = *p + *q;`	el arreglo a queda: 13 8 1
`*(p+1) = 0;`	el arreglo a queda: 13 0 1
`(*(q+1))++;`	el arreglo a queda: 13 0 2

Respuesta 33 Si, es posible, dado que el nombre del arreglo representa un puntero a la dirección base del arreglo. El significado de la expresión `*(a+i) = 6` es equivalente a `a[i] = 6`.

Respuesta 34 Inicializa en 0 las `n` componentes del arreglo pasado como argumento.

Respuesta 35 La primera invocación inicializa las 5 primeras componentes del arreglo `a` en 0. La segunda invocación inicializa las componentes b[5] a b[14] en 0. Notar que de ninguna manera la función `f` espera que siempre se le pase la dirección base de un arreglo. `x` se inicializa apuntando a la posición pasada como argumento y a partir de alli inicializa `n` componentes a 0.

Respuesta 36 El string `str2` contiene 6 caracteres: 'a', 'b', 'c', 'd', 'e'y el caracter nulo. El string destino `str1` sólo contiene lugar para 5 caracteres.

Respuesta 37 El string destino de la copia es un puntero que no está apuntando a un área para asignar los caracteres. Al no estar inicializado, apunta a cualquier lugar de la memoria, y si los caracteres son copiados alli, seguramente se producirá un error.

Respuesta 38 El programa crea dinámicamente en cada iteración un string cuya longitud depende del valor del índice `i`. El string es luego inicializado con `i` caracteres `á'` e impreso. Antes del fin de cada iteración, el lugar solicitado es retornado. Por supuesto, existen formas más directas y convenientes de lograr este objetivo.

la salida del programa es:

```
aa
aaa
aaaa
aaaaa
```

Respuesta 39 El problema surge porque se utilizó el signo `=` en la definición de la macro. El valor de `MAX` es entonces `=10`. Al expandir el `for`, el preprocesador obtiene:

```
for(cont ==10;cont > 0;cont--)...
```

y asume que se está tratando de comparar el valor de `cont` con 10, y `cont` no tiene valor asignado previamente.

Respuesta 40 Dos veces, dado que el `printf` es expandido a:

```
printf("%d\n",((cont++) * (cont++)));
```

D. Bibliotecas

Los programas en C pueden acceder de forma sencilla a los servicios del Sistema Operativo. Las invocaciones al sistema, esto es, las funciones que están disponibles a los programadores para acceder a los servicios del Sistemas Operativos, pueden ser invocados por los programas en C. Estas invocaciones al sistema proporcionan servicios para crear, leer y escribir en archivos, realizar operaciones matemáticas, entrada y salida de datos, etc.. Las bibliotecas (o también denominadas librerías) son colecciones de funciones interrelacionadas que se pueden utilizar de igual forma a las funciones programadas por el usuario, para incluirlas en el código se debe utilizar la directiva **#include**.

D.1. Funciones matemáticas: math.h

El encabezado **math.h** declara funciones matemáticas. Estas funciones modifican, en caso de existir un error, la variable global *errno* que determina el error ocurrido. Esta variable puede almacenar **EDOM** o **ERANGE**, que son constantes enteras con valor diferente de cero que se usan para señalar errores de dominio y de rango para las funciones.
Un *error de dominio* ocurre si un argumento está fuera del dominio sobre el que está definida la función. En caso de un error de dominio, `errno` toma el valor de **EDOM**; el valor regresado depende de la implementación. Un *error de rango* ocurre si el resultado de la función no se puede regresar como un double (lo excede). Si el resultado se desborda, la función regresa **HUGE_VAL** con el signo correcto, `errno` se hace **ERANGE**. Si el resultado es tan pequeño que no se puede representar (underflow), la función regresa cero: el que `errno` sea fijado a **ERANGE** depende de la implementación.

En la tabla siguiente, **x** e **y** son de tipo double y todas las funciones regresan double. Los ángulos para las funciones trigonométricas están representados en radianes.

FUNCIÓN	TAREA QUE REALIZA
`sin (x)`	seno de x.
`cos (x)`	coseno de x.
`tan (x)`	tangente de x.
`sinh (x)`	seno hiperbólico de x.
`cosh (x)`	coseno hiperbólico de x.
`tanh (x)`	tangente hiperbólica de x.
`exp (x)`	función exponencial.
`log (x)`	logaritmo natural con x>0, de lo contrario da a errno el valor EDOM.
`log10 (x)`	logaritmo base 10 con x>0.
`pow (x,y)`	x elevado a y. Da a errno el valor EDOM si x=0 y y<=0, o si x<0 y y no es un entero. Si lo retornado es de magnitud excesivamente grande errno toma valor ERANGE.
`sqrt (x)`	raíz cuadrada de x con x>=0.
`ceil (x)`	menor entero no menor que x, como double.
`floor (x)`	mayor entero no mayor que x, como double.
`fabs (x)`	valor absoluto.
`modf (x, double *ip)`	divide x en parte entera y fraccionaria, cada una con el mismo signo que x. almacena la parte entera en *ip, y regresa la parte fraccionaria.
`fmod (x, y)`	residuo de punto flotante de x dividido y, con el mismo signo que x. si y es cero, el resultado está definido por la implementación.

D.2. Funciones de cadenas de caracteres: string.h

Existen muchas funciones simples para strings ya definidas en esta biblioteca, para cálculo de longitud, concatenar cadenas, etc.. Para que puedan ser utilizadas se debe incluir el archivo de encabezamiento `strings.h`.

A continuación, se ven algunas de ellas:

int strlen(char *s)

Esta función retorna el número de caracteres en el string `s` (entero positivo), sin contar el caracter nulo. Cumple exactamente el papel de la función `longitud`, que se definió en la sección **??**.

Por ejemplo:

```
char s[] = "C es lindo";

printf("%d\n",strlen(s));
```

imprimirá:

```
10
```

void strcpy(char *s1,char *s2)

Esta función permite copiar strings y cumple exactamente el papel de la función `copia` que se definió en la sección **??**. Copia todos los caracteres apuntados por `s2` en el área apuntada por `s1`, hasta el caracter nulo que también es copiado. Debe haber sufiente espacio para los caracteres de `s2` en el área apuntada por `s1`.

Por ejemplo:

```
char a[100];
char b[] = "C es lindo";

strcpy(a,b);
printf("%s\n",a);
```

imprimirá:

```
C es lindo
```

void strcat(char *s1,char *s2)

Esta función concatena los caracteres apuntados por `s2` al final del string `s1`. Debe haber suficiente espacio en el área apuntada por `s1` para almacenar los caracteres de ambos strings.

Por ejemplo:

```
char a[100];

strcpy(a,"C es ");
strcat(a,"lindo");
printf("%s\n",a);
```

imprimirá:

```
C es lindo
```

int strcmp(char *s1,char *s2)

Esta función permite comparar lexicográficamente los strings `s1` y `s2`. Retorna 0 si ambos strings son iguales, un valor negativo si `s1` está lexicográficamente antes de `s2`, y uno positivo si `s1` está después de `s2`.

Por ejemplo:

`strcmp("hola","hola")`	retorna 0
`strcmp("hola","hello")`	retorna un valor positivo
`strcmp("hello","hola")`	retorna un valor negativo

Existen muchas más funciones de biblioteca para strings. Es conveniente verificar primero si la tarea que uno desea hacer no puede ser simplificada usando alguna otra función ya definida en la biblioteca.

D.3. Funciones de utilería: stdlib.h

void exit (int status)
exit ocasiona la terminación normal del programa dependiendo del valor del parámetro **status**. Las funciones atexit se llaman en orden inverso del registrado, los archivos abiertos se vacían, los flujos abiertos se cierran, y el control se regresa al entorno. Cómo se regrese **status** al entorno depende de la implementación, pero cero indica cuando la terminación tiene éxito y un valor distinto indica algún tipo de error. Se pueden utilizar también los valores **EXIT_SUCCESS** y **EXIT_FAILURE**.

void abort (void)
abort ocasiona que el programa termine anormalmente.

D.4. Funciones de entrada y salida con archivos: stdio.h

Un flujo (stream) es una fuente de datos que puede estar asociada con un disco o un periférico. Esta biblioteca maneja flujos o stream de *texto* o *binarios*. Un flujo de texto es una secuencia de líneas, cada línea tiene cero o más caracteres y está terminada por `'\n'`. Un flujo binario es una secuencia de bytes no procesados que representan datos internos escritos en lenguaje binario. Un flujo se conecta a un archivo o dispositivo al abrirlo, la conexión se rompe cerrando dicho flujo.
Las siguientes funciones tratan con operaciones sobre archivos. El tipo **size_t** es el tipo entero sin signo producido por el operador **sizeof**.

FILE *fopen (const char * filename, const char * mode)
fopen abre el archivo nombrado, y regresa un flujo, o NULL si falla el intento. Los valores legítimos de mode incluyen:

VALORES DE MODE	TAREA QUE REALIZA
`"r"`	abre archivo de texto para lectura
`"w"`	crea archivo de texto para escritura; descarta el contenido previo si existe
`"a"`	agrega; abre o crea un archivo para escribir al final
`"r+"`	abre archivo para actualización (esto es, lectura o escritura)
`"w+"`	crea archivo de texto para actualización; descarta cualquier contenido previo si existe
`"a+"`	agrega; abre o crea archivo de texto para actualización, escribiendo al final

El modo de actualización permite la lectura y escritura del mismo archivo; entre una lectura y una escritura debe llamarse a fflush o a una función de posición. Si el modo incluye **b** después de la letra inicial, como en `"rb"` o "w+b", indica que el archivo es binario. Los nombres de archivo están limitados a FILENAME_MAX caracteres. Pueden ser abiertos hasta un máximo de archivos determinado por la constante FOPEN_MAX.

int fflush (FILE *stream)
Para un flujo de salida, **fflush** ocasiona que sea escrito cualquier dato con uso de buffer que hasta ese momento no ha sido escrito; para un flujo de entrada, el efecto esta indefinido. Regresa EOF en caso de un error de escritura, y cero en caso contrario.

int fclose (FILE *stream)
fclose descarga cualquier dato no escrito de stream, descarta cualquier buffer de entrada no leído, libera cualquier buffer asignado automáticamente, después cierra el flujo. Regresa EOF si ocurre cualquier error y cero en caso contrario.

int remove (const char *filename)

remove remueve el archivo nombrado, de modo que en un intento posterior de abrirlo fallará. Regresa un valor diferente de cero si el intento falla.

int rename (const char *oldname, const char *newname)
rename cambia el nombre de un archivo; regresa un valor distinto de cero si el intento falla.

FILE *tmpfile (void)
tmpfile crea un archivo temporal con modo "wb+"que será removido automáticamente cuando se cierre o cuando el programa termine normalmente. tmpfile regresa un flujo, o NULL si no puede crear el archivo.

char *tmpnam (char s [L_tmpnam])
tmpnam (NULL) crea una cadena que no es el nombre de un archivo existente y regresa un apuntador a un arreglo estático interno. tmpname(s) almacena la cadena en s y también la regresa como el valor de una función; s debe tener espacio suficiente para al menos L_tmpnam caracteres. tmpnam genera un nombre diferente cada vez que se invoca; cuando más están garantizados TMP_MAX diferentes nombres durante la ejecución del programa. Nótese que tmpnam crea un nombre, no un archivo.

D.4.1. Lectura y escritura en archivos ASCII o de textos

Escritura de un carácter en un archivo.
Existen dos funciones definidas en **stdio.h** para escribir un carácter en el archivo. Ambas realizan la misma función y ambas se utilizan indistintamente. La duplicidad de definición es necesaria para preservar la compatibilidad con versiones antiguas de C. Los prototipos de ambas funciones son:

int putc(int c, FILE *nombre_archivo);

int fputc(int c, FILE * nombre_archivo);

Donde nombre_archivo recoge la dirección que ha devuelto la función fopen. El archivo debe haber sido abierto para escritura y en formato texto, y donde la variable c es el carácter que se va a escribir. Si la operación de escritura se realiza con éxito, la función devuelve el mismo carácter escrito. Lectura de un carácter desde un archivo. De manera análoga a las funciones de escritura, existen también funciones de lectura de caracteres desde un archivo. De nuevo hay dos funciones equivalentes, cuyos prototipos son:

int fgetc(FILE *nombre_archivo);

int getc(FILE * nombre_archivo);

Que reciben como parámetro el puntero devuelto por la función fopen al abrir el archivo y devuelven el carácter, de nuevo como un entero. El archivo debe haber sido abierto para lectura y en formato texto. Cuando ha llegado al final del archivo, la función fgetc, o getc, devuelve una marca de fin de archivo que se codifica como EOF.

Lectura y escritura de una cadena de caracteres.
Las funciones **fputs** y **fgets** escriben y leen, respectivamente, cadenas de caracteres sobre archivos de disco. Sus prototipos son:

int fputs(const char *s, FILE *nombre_archivo);

char *fgets(char *s, int n, FILE * nombre_archivo);

La función fputs escribe la cadena s en el archivo indicado por el puntero nombre_archivo. Si la operación ha sido correcta, devuelve un valor no negativo. El archivo debe haber sido abierto en formato texto y para escritura o para lectura, dependiendo de la función que se emplee. La función fgets lee del archivo indicado por el puntero nombre_archivo una cadena de caracteres. Lee los caracteres desde el inicio hasta un total de n, que es el valor que recibe como segundo parámetro. Si antes del carácter n-ésimo ha terminado la cadena, también termina la lectura y cierra la cadena con un carácter nulo.

Lectura y escritura con formato

Las funciones fprintf y fscanf de entrada y salida de datos por disco tienen un uso semejante a las funciones printf y scanf, de entrada y salida por consola. Sus prototipos son:

int fprintf(FILE *F, const char *cadena_de_control, ...);

int fscanf(FILE *F, const char *cadena_de_control, ...);

Donde F es el puntero a archivo que devuelve la función fopen. Los demás argumentos de estas dos funciones son los mismos que en las funciones de entrada y salida por consola. La función fscanf devuelve el carácter EOF si ha llegado al final del archivo. El archivo debe haber sido abierto en formato texto, ya sea para escritura o para lectura dependiendo ésto de la función que se emplee.

D.4.2. Lectura y escritura en archivos binarios.

En todas las funciones anteriores se ha requerido que la apertura del fichero o archivo se hiciera en formato texto, para hacer uso de las funciones de escritura y lectura en archivos binarios el archivo será abierto en formato binario. Las funciones siguientes permiten la lectura o escritura de cualquier tipo de dato. Los prototipos son los siguientes:

size_t fread (void *buffer, size_t n_bytes, size_t contador, FILE *nombre_archivo);

size_t fwrite (const void *buffer, size_t n_bytes, size_t contador, FILE *nombre_archivo);

Donde buffer es un puntero a la región de memoria donde se van a escribir los datos leídos en el archivo, o el lugar donde están los datos que se desean escribir en el archivo. Habitualmente será la dirección de una variable. n_bytes es el número de bytes que ocupa cada dato que se va a leer o grabar, y contador indica el número de datos de ese tamaño que se van a leer o grabar. El último parámetro es el de la dirección que devuelve la función fopen cuando se abre el archivo. Ambas funciones devuelven el número de elementos escritos o leídos. Ese valor debe ser el mismo que el de la variable contador, a menos que haya ocurrido un error. Estas dos funciones son útiles para leer y escribir cualquier tipo de información. Es habitual utilizarlas junto con el operador *sizeof*, para determinar así la longitud (n_bytes) de cada elemento a leer o escribir.