Lenguaje C
Tutorial

Fund. de la Informática - Ing. Electrónica
Introducción a la Programación - T.U.G.
Introducción a la Programación - T.U.M.
Introducción a la Programación - Prof. en Tec. Electrónica

Área de Servicios
Departamento de Informática
Universidad Nacional de San Luis

Índice

1. Introducción **3**

2. El primer programa **3**

3. Un programa más complejo **4**

4. Tipos numéricos **6**
4.1. El tipo entero . . . 6
4.2. El tipo caracter . . . 7
4.3. Constantes enteras . . . 8
4.4. Combinando enteros . . . 9
4.5. El tipo flotante . . . 10
4.6. Combinando flotantes con enteros . . . 10
4.7. Conversión de tipos . . . 10

5. Operadores **11**
5.1. Operadores aritméticos . . . 11
5.2. Operadores relacionales . . . 12
5.3. Operadores lógicos . . . 13
5.4. Operadores a nivel de bits . . . 13
5.5. Operadores de asignación . . . 14
5.6. Operadores de incremento y decremento . . . 15
5.7. Operador condicional . . . 16

6. Control de secuencia **16**
6.1. La sentencia de selección `if` . . . 16
6.2. La sentencia de selección `switch` . . . 17
6.3. La sentencia de iteración `while` . . . 18
6.4. La sentencia de iteración `do while` . . . 19
6.5. La sentencia de iteración `for` . . . 19
6.6. Las sentencias `break` y `continue` . . . 21

7. Objetos de datos estructurados **23**
7.1. Arreglos . . . 23
7.2. Estructuras . . . 24
7.3. Uniones . . . 25
7.4. Combinación . . . 26

8. Punteros **27**
8.1. Punteros como parámetros . . . 31
8.2. Aritmética de punteros . . . 35
8.3. Punteros y arreglos . . . 35

9. Strings **42**
9.1. Definiendo funciones para strings . . . 43
9.2. Funciones de biblioteca . . . 45

10. Entrada y salida estándar **47**
10.1. Entrada y Salida Básica . . . 47
10.2. Entrada y Salida Formateada . . . 47

11. Archivos **49**
11.1. Archivos ASCII y Binarios 51
11.2. Trabajo con los archivos 51
11.2.1. Archivos ASCII 51
11.2.2. Archivos Binarios 52
11.3. Ejemplos 53

12. Administración dinámica de memoria **56**

13. Parámetros del programa **58**

14. Declaraciones de tipos **59**

15. Estructura del programa **60**
15.1. Clases de almacenamiento 60
15.2. Alcance 60

16. El preprocesador **62**

A. Precedencia y asociatividad de las operaciones **65**

B. Especificadores de formato **66**

C. Respuestas **67**

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 dia. De hecho, el disño 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 facil 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. La función `main`, cuya definición aparece en este pequeño programa, es una función especial: es el punto donde comienza le ejecución del programa. Desde este punto de vista se puede comparar al programa principal de un programa Pascal. Note que aquí se utilizan llaves ({ y }) en forma similar a la forma en que en Pascal se utilizan las sentencias `begin` y `end`.

El encabezamiento de la definición de la función:

```
  int main()
```

indica que la función `main` no tiene 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?. En Pascal no existe equivalencia a estos conceptos. Los argumentos al programa principal, o función `main` serán tratados en una sección posterior. El valor retornado, un entero en este caso, es una valor que es pasado por el programa 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 definiremos 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 (más detalles más adelante), en la que se indica que se deben incluir las definiciones de la biblioteca de entrada/salida estándar (**st**an**d**ard **i**nput **o**utput). 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\nuna ");
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.
```

3. Un programa más complejo

A continuación se presenta un programa más elaborado.

```
#include <stdio.h>

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

int main() {
  int result;
```

```
    result = doble(5);
    printf("el doble de 5 es %d\n",result);
    return 0;
}
```

Este programa incluye dos funciones. La función `main` y la función `doble`. La especificación de la función `main` ya fue discutida en la sección anterior.

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 a diferencia de Pascal, 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 entero y toma como argumento un entero a través del parámetro formal `x`.

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

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

Esta es una expresión de asignación que asigna a la variable `result` el valor entero retornado por la función `doble` al ser invocada con el valor 5.

En este punto 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;
```

Es la misma sentencia `return` que fue explicada en el ejemplo anterior. El valor del parámetro formal `x` es multiplicado por 2 y ese resultado es retornado por la función.

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

A continuación el valor es impreso a través de la función `printf`:

```
printf("el doble de 5 es %d\n",result);
```

obteniéndose en la pantalla:

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

Notar que la fución `printf` recibe un argumento adicional, que es el valor de la variable `result`. El primer string de la función `printf` se conoce como *string de formato*, y su función es 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. Si aparece el 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. Por ejemplo, el especificador `%d` indica que el valor de la siguiente variable, `result` en este caso, será impreso como un entero decimal. Esta es la razón por la que 10 es impreso.

Se pueden indicar más especificadores de formato, por ejemplo:

```
int i,j;

i = 1;
j = 2;
printf("el valor de i es %d y el de j es %d\n",i,j);
```

imprimirá

```
el valor de i es 1 y el de j es 2
```

Pregunta 3 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;
}
```

4. Tipos numéricos

4.1. El tipo entero

El tipo entero básico de C es el que hemos 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 de 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 que estaremos seguros 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 4 ¿Es posible que en una implementación particular, un `short`, un `long` y un `int` tengan todos el mismo tamaño?

4.2. El tipo caracter

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

```
char c;
```

le podemos asignar caracteres:

```
c = '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:

```
c = 65;
```

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

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

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

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

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

i = c * 2;
```

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

Pregunta 5 ¿Cuál es la salida del siguiente trozo de programa?

```
char c = 'A';

printf("%c %c %c",c,c+1,c+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 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 c;

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

imprimirá:

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

4.3. 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, supongamos que en nuestra 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 OK */
```

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

```
unsigned y;

y = 455u;
```

Pregunta 6 ¿Cuál es el valor asignado a las variables enteras?

```
int a,b;

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

Pregunta 7 Identifique el error en el siguiente trozo de programa:

```
int a;

a = 08 + 02;
```

Pregunta 8 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;
```

4.4. 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 tenemos 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. Observe 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 9 ¿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
```

4.5. 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 */
```

4.6. Combinando flotantes con enteros

Se debe tener cuidado cuando se mezclan enteros con flotantes dado que se puede 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 comprenderá entonces que se desea realizar una división de flotantes, por ejemplo:

```
float f;

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

4.7. Conversión de tipos

Tal como se ha visto en la sección 4.4 y en la 4.6, 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). Claramente 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 4.6, 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 10 ¿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 11 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 12 ¿Cuáles son los valores asignados?.

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

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

5. 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.

5.1. 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 4.4 y 4.6.

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;
```

5.2. 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 13 ¿Cuál es el valor de las siguientes expresiones?.

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

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

5.3. Operadores lógicos

Los operadores lógicos son los siguientes:

operador	descripción
`&&`	and
`||`	or
`!`	not

Estos operadores tienen una característica muy interesante, y es el hecho que 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 14 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)
```

5.4. 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 15 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 16 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?.

5.5. Operadores de asignación

En C la asignación es una operación y no una sentencia como en Pascal. Si bien lo hemos 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;
```

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 notable 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 17 Escriba en versión extendida normal las siguientes expresiones compactas:

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

5.6. 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 18 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--;
```

5.7. 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 19 ¿ 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--;
```

6. Control de secuencia

6.1. 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 ganamos 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 ganamos 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 ganamos nada\n");
else
  printf("ganamos %d pesos\n",ganancia);
```

Pregunta 20 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 21 El siguiente programa parece empecinado a que nuestro balance sea cero. Explique la razón:

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

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

El programa imprime:

```
nuestro balance es 0
```

6.2. La sentencia de selección `switch`

El `switch` es una sentencia de selección con múltiples ramas similar al case de Pascal. 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.

Si bien el comportamiento es similar al case de Pascal, hay una diferencia fundamental: 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 22 ¿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 encontrada 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).

6.3. 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 4.2).

```
/* 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.

6.4. 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;
}
```

6.5. 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 las 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 cuerpoy 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. Note 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
```

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

Pregunta 23 Ejecute la función `factorial` para `n` igual a 0,1,2 y 4.

Pregunta 24 Reescriba el ejemplo dado en la sección 6.3 usando for.

Pregunta 25 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;
}
```

6.6. Las sentencias `break` y `continue`

La sentencia `break` tiene dos usos. Uno ya fue explicado en la sección 6.2 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. Note que la sentencia `i++` es ejecutada dado que siempre se ejecuta el 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 26 Determine cual es el objetivo del siguiente programa:

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

int main() {
  int i,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 27 Reesciba el programa anterior sin usar `break` ni `continue`.

7. Objetos de datos estructurados

7.1. 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 subscriptos 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 una arreglo de 10 componentes de tipo caracter, de las cuales sólo la primera y la segunda han sido inicializadas, con los valores `á'` y `'b'` respectivamente.

7.2. 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 deseáramos mantener en un programa los datos de un empleado, primero tendríamos que identificar los atributos de interés, como ser nombre, domicilio, sueldo, etc.

Una estructura es similar al registro de Pascal. 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, podríamos 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.

7.3. 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.

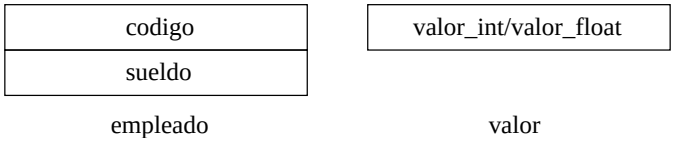

Figura 1: representación de estructuras y uniones

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 28 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;
    }
```

7.4. 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 29 ¿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 30 ¿Es posible utilizar el campo `cod_pais` para un producto nacional?

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 definimos 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 2.

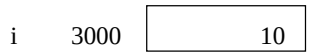

Figura 2: variable en memoria

Un puntero a variables de tipo entero se define como sigue:

```
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 3 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**.

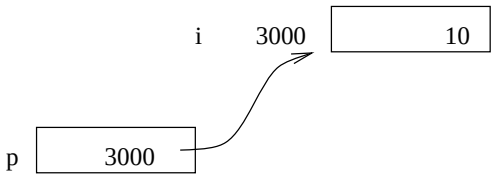

Figura 3: la variable es apuntada por el puntero

Podemos 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 describimos.

`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.
Supongamos el siguiente ejemplo:

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

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; /* declaramos 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 sería:

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

```
punt=NULL;
punt=(int *) 285395; /* asignaci\'on de direccion */
/* absoluta */
```

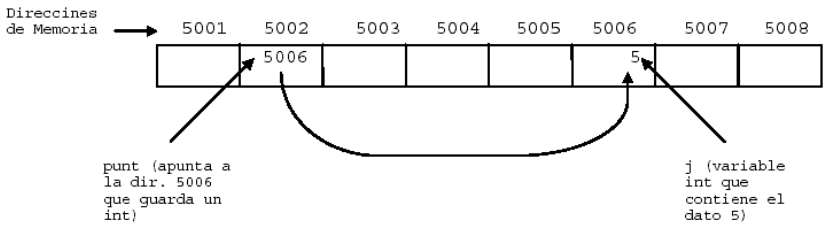

Figura 4: Punteros

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 declaramos dos punteros y una variable entera, y hacemos 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 pasamos 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. Sabemos que cuando se pasa como parámetro una variable a una función, esta no logra modificar el valor de la variable, eso es dado 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++;
}

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

[4] El tipo de valor retornado por la función se define como **void**, esto significa que en realidad no devuelve ningún valor y por lo tanto es equivalente a un procedimiento de Pascal

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 5).

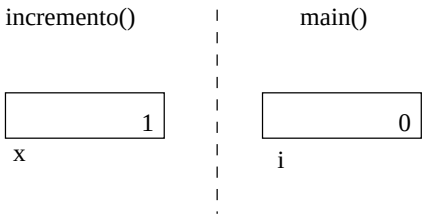

Figura 5: pasaje por valor

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)++;
}

int main() {
  int i = 0;

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

Note que se pasa la dirección de la variable, y el parámetro formal se define como un puntero, ya que éste recibirá una dirección. Note 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 6.

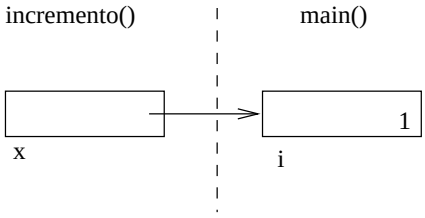

Figura 6: pasaje por valor de una dirección

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

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

```
#include <stdio.h>
 void swap();   /* Funci\'on para el intercambio */

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

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

Nuestro 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 nuestra función lo que se debe hacer es intercambiar los valores. Como los parámetros son punteros, entonces debemos 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 es:

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

Con esto logramos nuestro objetivo, la salida al programa es:
a=2 b=4
a=4 b=2

La figura 7 muestra la evolución de los valores durante la ejecución de la función **swap**.

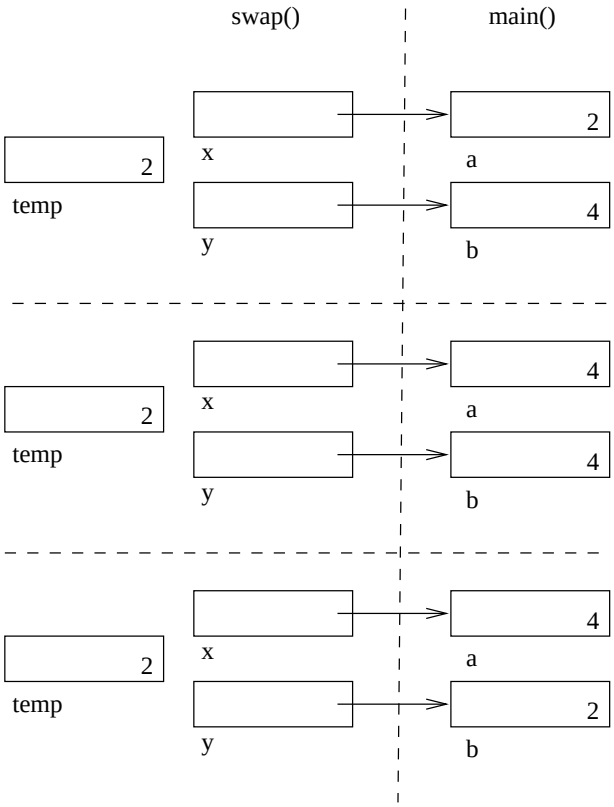

Figura 7: pasos de la ejecución de la función swap

Ejercicio: Crear un programa que lea dos números flotantes, 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);
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 utilizamos 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);
}
```

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. 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 *q,dato;
q=&dato;
q++;
```

8.3. 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 dijimos 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 8).

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];
```

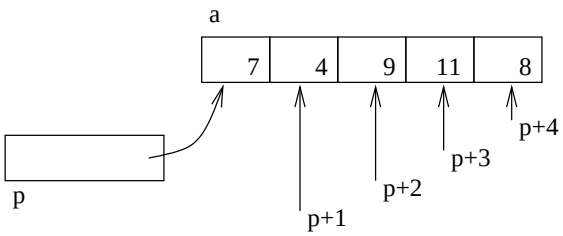

Figura 8: sumando constantes a un puntero

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. Consideremos el siguiente ejemplo, en el que invocamos 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 estamos pasando 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;
}
```

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;
}
```

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

Para C, ambas declaraciones son absolutamente equivalentes, por lo que independientemente de como se declare el argumento, tanto los accesos como punteros o con índices son equivalentes. Es decir, son válidos los siguientes usos:

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

  *(p+1) = 3;               *(x+1) = 3;
  p[2] = 4;                 x[2] = 4;
}                         }
```

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 d\'ecimo elemento es : %d",*(p+9));
```

para referirnos 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 queremos, porque?, 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("\np 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 asumimos 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;
}

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

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

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. Todos los ejemplos dados anteriormente son a título informativo, pues no están incluidos en esta materia.

Un ejercicio gráfico

El ejercicio de la figura 9 pretende mostrar gráficamente que ocurre con las variables y punteros durante la creación y asignación.

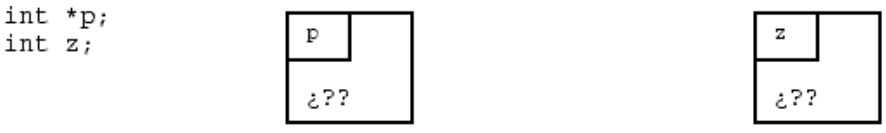

Figura 9: Creación

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.

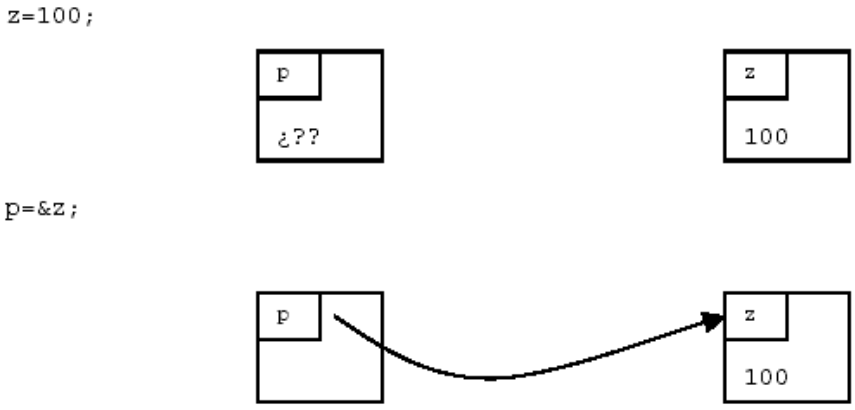

Figura 10: Asignación

Despues de las asignaciones de la figura 10, 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.
Supongamos ahora que se hace la siguiente asignación, ver figura 11,

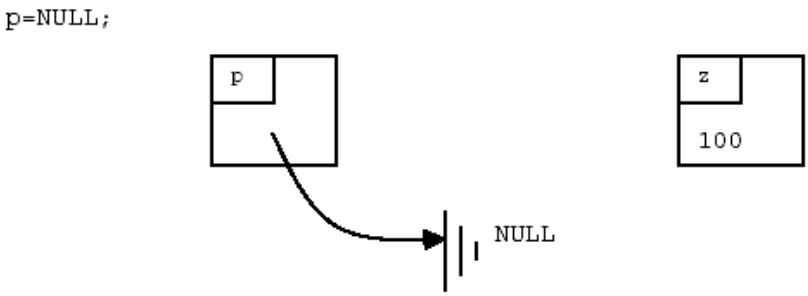

Figura 11: Valor NULL

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 12:
Después de esta asignación, el contenido de **p** es igual a 150, y **z** aún mantiene el valor 100.

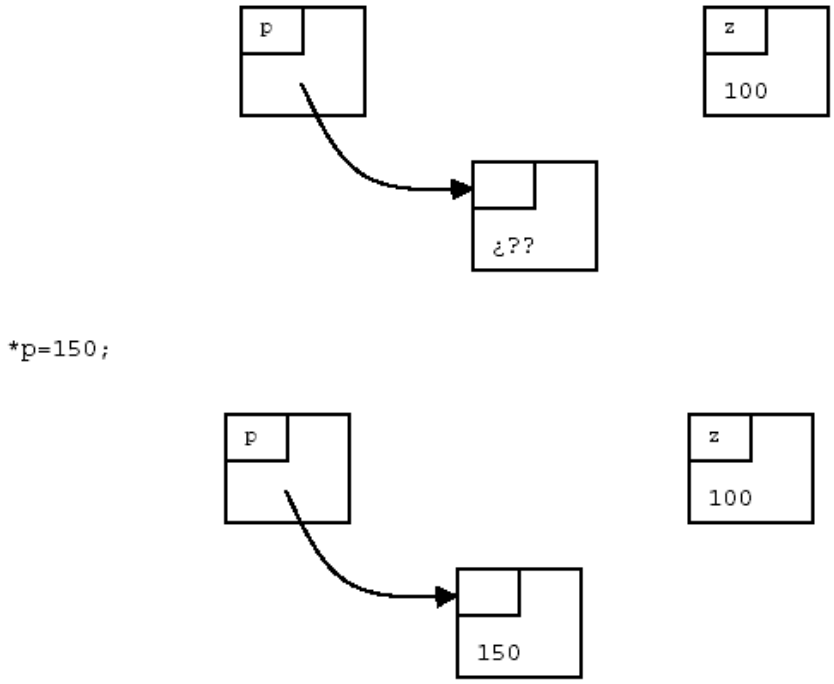

Figura 12: Reserva de memoria y asignación de valor

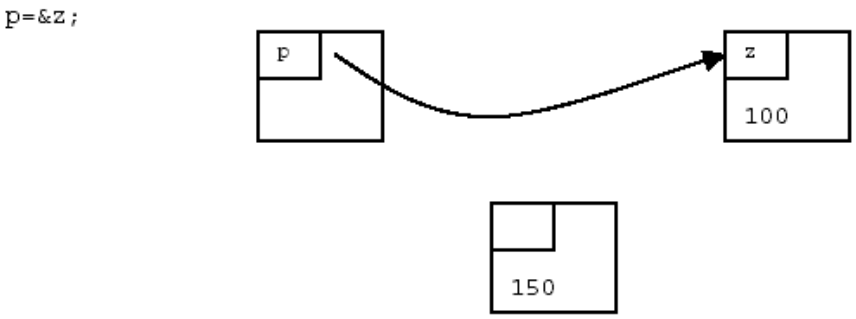

Figura 13: Uso del espacio de memoria

Si ahora hacemos nuevamente (ver figura 13):

Con esto, en nuestra jerga diríamos que: "se nos 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 14.
Ahora, tanto **z** como el contenido de **p** tienen el valor 200. Finalmente, si:

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

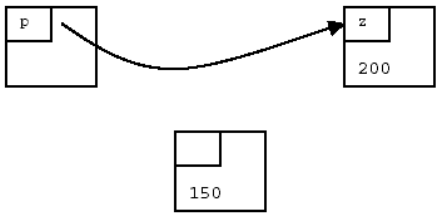

Figura 14: Uso del espacio de memoria

imprimiría en pantalla:

```
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 esta 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.

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 como sigue:

```
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. Note que un caracter adicional se necesita para indicar el fin del string. Una notación más compacta para inicializar un string es la siguiente:

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

que tiene exactamente el mismo efecto que la anterior.

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";
```

El nombre de un arreglo es además un puntero a la dirección base del arreglo (recuerde la sección 8.3). Las siguientes declaraciones son válidas, aunque hay una diferencia sutil en términos de su representación de memoria, tal como se puede observar en la figura 15.

```
char str1[] = "hola";
char *str2 = "hola";
```

En el segundo caso, tenemos un lugar en la memoria para almacenar los caracteres y un puntero real a caracteres. En ambos casos, las notaciones con índices y con punteros son válidas. Es decir, que todas las expresiones siguientes son válidas:

`str1[2]`	tiene valor 'l'
`*(str1+2)`	equivalente a la anterior
`str2[3]`	tiene valor á'
`*(str2+3)`	equivalente a la anterior

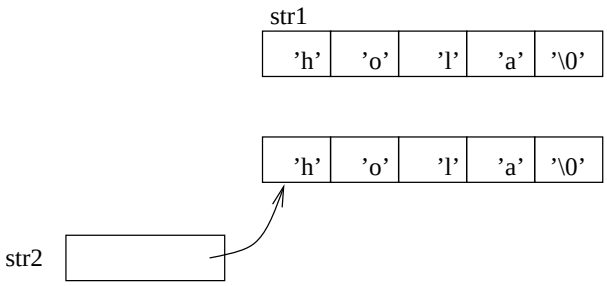

Figura 15: representación en memoria

Una diferencia importante, sin embargo, es que en el segundo caso se está definiendo un puntero real que puede ser modificado. En el primer caso, el puntero es el nombre del arreglo, y por esa razón, no se puede modificar. Es decir que:

`str1++` no está permitido
`str2++` es legal y avanza el puntero una posición

9.1. Definiendo funciones para strings

Dado que la longitud de los strings es variable, podemos definir una función que calcule cuantos caracteres tiene un string. La función la denominaremos `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:

```
char str1[10] = "abc";
char *str2 = "abcde";

printf("%d\n",longitud(str1));  /* imprime 3 */
printf("%d\n",longitud(str2));  /* imprime 5 */
```

Una posible definición de esta función es la siguiente:

```
int longitud(char s[]) {
  int i;

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

Esta función incrementa un contador `i`, y mientras en la posición `i` no aparezca el caracter nulo, continúa incrementandolo.

Note 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 que 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, debemos copiar todos los caracteres, uno por uno. Para ello, podemos definir una función `copia`, que toma el string destino como primer argumento, el string fuente como segundo, y realiza la copia de todos los caracteres incluído el caracter nulo.

```
char str1[6];
char str2[] = "abc";

copia(str1,str2);
printf("%s",str1); /* imprime abc */
```

Una posible definición de esta función es la siguiente:

```
void copia(char s1[],char s2[]) {
  int i;

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

Esta función utiliza un índice para recorrer los arreglos, y mientras no encuentra el caracter nulo, asigna el i-ésimo caracter de `s2` en `s1`. Luego el caracter nulo es copiado fuera de la iteración.

Si bien esta función es correcta, no está escrita en la forma más compacta y eficiente posible. La forma normal en la que esta función se define en C es la siguiente:

```
void copia(char *s1,char *s2) {
  while (*s1++ = *s2++);
}
```

El código de esta función es extremadamente compacto y eficiente, aunque puede resultar un poco intimidante la primera vez que se ve. Los valores de las variables y de los parámetros formales en el momento de la invocación se muestran en la primera parte de la figura 16.

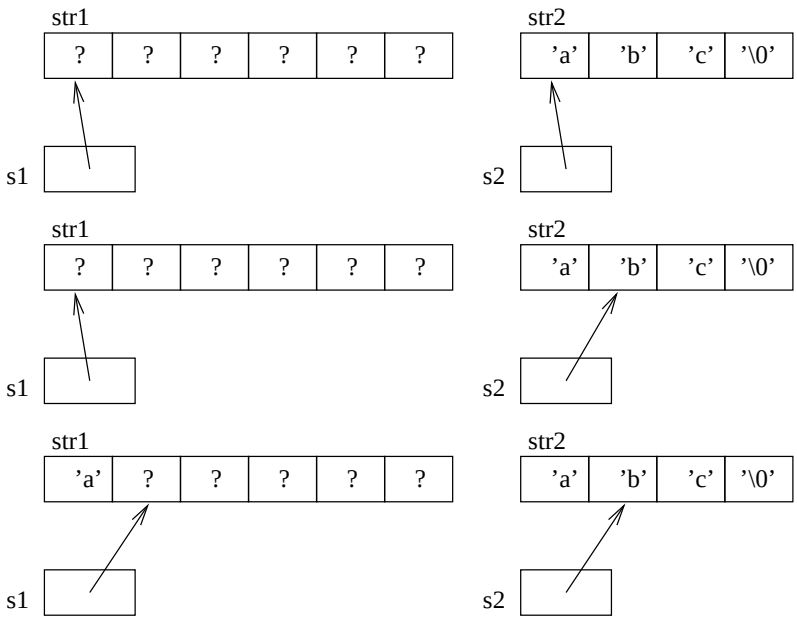

Figura 16: primera iteración

Note que la iteración no tiene cuerpo, es decir, que la condición será continuamente evaluada hasta que de resultado falso (o sea igual a 0). La condición es una asignación:

```
*s1++ = *s2++;
```

asi que el valor asignado será utilizado para determinar si la condición es verdadera o falsa. Si el valor asignado es 0 (o el caracter nulo) la condición será falsa. En cualquier otro caso será verdadera.

En la asignación, el lado derecho es considerado primero:

```
*s2++
```

El operador `++` es ejecutado primero, por lo que el puntero `s2` avanza a la siguiente posición. Sin embargo, es una operación de post-incremento, por lo que el resultado es el valor del puntero antes de realizar la operación. Este valor es desreferenciado por el operador `*`. En el ejemplo, la primera vez que la expresión se evalúa, `s2` pasará a apuntar a `str2[1]`, y el caracter obtenido será `á'` (ver la segunda parte de la figura 16).

En el lado izquierdo de la asignación tenemos:

```
*s1++
```

donde el proceso es el mismo, `s1` es avanzado una posición, pero en la posición en la que estaba apuntado se asigna el caracter `á'` (ver la tercera parte de la figura 16).

El proceso continúa hasta que todos los caracteres son copiados, incluyendo el caracter nulo. Al copiar este caracter la condición se vuelve falsa, y el proceso termina tal como se muestra en la figura 17.

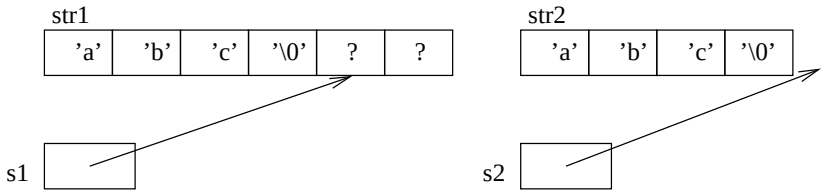

Figura 17: justo antes de retornar de copiar

Note que al final del proceso el puntero `s2` queda apuntando fuera del string. Eso no consituye un error mientras no se intente acceder a esa posición de la memoria.

Pregunta 36 ¿Dónde está el error en el siguiente trozo de programa?

```
char str1[5];
char str2[] = "abcde";

copia(str1,str2);
```

Pregunta 37 ¿Dónde está el error en el siguiente trozo de programa?

```
char *str1;
char *str2 = "abcde";

copia(str1,str2);
```

9.2. Funciones de biblioteca

Existen muchas funciones para strings ya definidas en la biblioteca. Para que puedan ser utilizadas se debe incluir el archivo de encabezamiento `strings.h`. Veremos ahora algunas de ellas:

- `strlen(char *s)`

  Esta función retorna el número de caracteres en el string `s`, sin contar el caracter nulo. Cumple exactamente el papel de la función `longitud`, que se definió en la sección 9.1.

  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 9.1. 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 despues 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.

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:

```
main() {
  int ch;

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

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:

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

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 ingresamos 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 Apéndice 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 importarnos), 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(8);
}
```

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 artículos 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);
}
```

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);
}
```

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);
}
```

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);
}
```

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);
}
```

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, podemos 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 requerimos 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 estamos seguros 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).

Supongamos 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 dinamicamente 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 7.2):

```
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 ese el tipo de argumento que espera la función `free`.

Pregunta 38 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 18.

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 lo compilamos, le damos el nombre `programa` a la versión ejecutable, obtendremos como resultado de la invocación mostrada al comienzo:

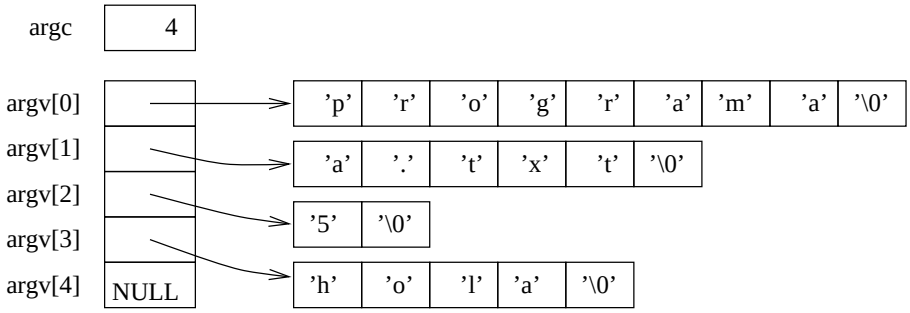

Figura 18: parámetros del programa

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

Note 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`.

Note 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 podemos 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;
```

[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.

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 7.2).

15. Estructura del programa

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

15.1. Clases de almacenamiento

Hemos visto que un programa consiste de un conjunto de funciones. Todas las variables que hemos 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):

```
#include <stdio.h>

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

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

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:

[7] ¡Use la regla!

```
void f() {
  int i;

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

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

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);
}

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

void main() {
  a = 12;
  f();
}
```

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.

Consideremos 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`.

Note la diferencia entre definición y declaración. Una declaración especifica los atributos solamente, mientras que una definición hace la misma cosa, pero también asigna memoria. Note 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`. Note 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` que hemos usado 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;
}
```

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;
}
```

y recién entonces es compilado. Note 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 39 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 40 ¿Cuantas 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 4.7.

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. Note 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	3

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`. Note 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 No, dado que el primer parámetro es un entero, y se pasa un entero por valor (`a`). El segundo parámetro es un puntero a entero, y al pasar la dirección de `b` se está pasando un puntero a entero. El tercer parámetro también es un puntero a entero, y se está pasando `p`, que es un puntero a entero que está apuntando en este momento a la variable `a`.

Respuesta 33 El programa imprime:

```
1 1 - 1 0
2 1 - 2 0
3 1 - 3 0
```

Respuesta 34 La representación gráfica se presenta en la figura 19.

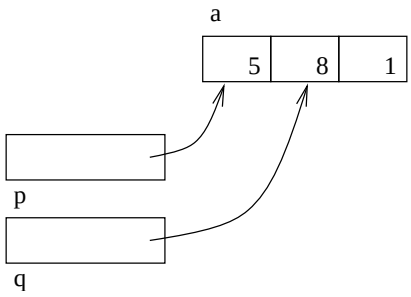

Figura 19: arreglo y punteros

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 35 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 36 Inicializa en 0 las `n` componentes del arreglo pasado como argumento.

Respuesta 37 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. Note 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 38 El puntero `p` apunta inicialmente a la primera componente del arreglo `a` y el puntero `q` a la tercera. El efecto de las expresiones es el siguiente:

asignación	comentario
`p++;`	p pasa a apuntar a la segunda componente
`i = *p - *q;`	asigna a i el valor 1
`i = a[q - p];`	asigna a i el valor 4
`a[2] = *(q + 1);`	el arreglo queda: 6 4 2 2

Respuesta 39 Ambas retornan 4. La pregunta tiende a confundir con respecto al tamaño de las componentes de los arreglos. La resta no es una resta de los punteros en valor absoluto, es decir, en término de sus direcciones de memoria. Fue definida como el número de componentes entre los dos punteros, es decir, que el tamaño de las componentes no influye.

Respuesta 40 La razón es que el compilador no puede determinar el tipo del objeto de datos que se desea referenciar a través de la expresión `*p`. Es distinto el caso si tenemos, por ejemplo:

```
int *r;
```

debido a que al encontrar la expresión conoce el tipo de datos al que apunta `r` (un entero), y está en condiciones de pasarlo como argumento para la función `printf`.

Respuesta 41 El string `str2` contiene 6 caracteres: á', 'b', 'c', 'd', é'y el caracter nulo. El string destino `str1` sólo contiene lugar para 5 caracteres.

Respuesta 42 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 43 Se debe a que la función `scanf` debe asignar los valores leídos desde la entrada estándar a esas variables. Como el pasaje de parámetros en C es por valor, si se pasaran las variables y no sus direcciones, no podrían ser modificadas.

Respuesta 44 El programa lee uno a uno los caracteres del archivo `a.txt` y los imprime uno a uno en el archivo `nuevo.txt`. El efecto es que el archivo `a.txt` es copiado en el archivo `nuevo.txt`

Respuesta 45 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 46 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 47 Dos veces, dado que el `printf` es expandido a:

```
printf("%d\n",((cont++) * (cont++)));
```