

TEORÍA Y PROBLEMAS

PUNTEROS EN C

FREDY CUENCA, PhD

Copyright © 2022 Fredy Cuenca

Todos los derechos reservados. Ninguna parte de este libro puede ser reproducida de ninguna manera sin el permiso escrito del autor.

Primera edición en Junio del 2022

ISBN 978-1-4583-0679-1

*A los que creen
aunque se equivoquen*

CONTENIDO

1. DATOS ATÓMICOS y DATOS AGREGADOS	1
VARIABLES.....	2
DECLARACIÓN DE VARIABLES.....	2
ASIGNACIÓN DE VARIABLES	3
UNA ASIGNACIÓN NO ES UNA ECUACIÓN	4
ASIGNACIÓN MÚLTIPLE.....	5
MOSTRAR VARIABLES EN PANTALLA.....	5
EJECUCIÓN DE UN PROGRAMA.....	7
DEL TECLADO A LA VARIABLE.....	9
OPERADORES.....	10
ARREGLOS DE VARIABLES.....	13
ARREGLOS	13
MATRICES.....	15
CADENAS	17
VARIABLES DE TIPOS DEFINIDOS POR USUARIO	21
ESTRUCTURAS DE DATOS.....	21
ENUMERACIONES.....	24
CAMPOS DE BITS	26
PROBLEMAS RESUELTOS	29
PROBLEMAS PROPUESTOS	38
2. PROGRAMACIÓN ESTRUCTURADA	39
SENTENCIAS CONDICIONALES.....	40
SENTENCIA IF-ELSE	40
SENTENCIA SWITCH.....	43
OPERADOR TERNARIO.....	45
SENTENCIAS ITERATIVAS	47
BUCLE FOR.....	47
BUCLE WHILE.....	52
BUCLE DO-WHILE	54
EL VIEJO GOTO	56

PROBLEMAS RESUELTOS	58
PROBLEMAS PROPUESTOS	73
3. REPRESENTACIÓN INTERNA DE DATOS ATÓMICOS	77
REPRESENTACIÓN INTERNA DE ENTEROS.....	78
ENTEROS SIN SIGNO.....	78
ENTEROS CON SIGNO.....	79
REPRESENTACIÓN INTERNA DE NÚMEROS DECIMALES	82
NÚMEROS DE COMA FLOTANTE DE PRECISIÓN SIMPLE, <code>float</code>	82
NÚMEROS DE COMA FLOTANTE DE PRECISIÓN DOBLE, <code>double</code>	86
REPRESENTACIÓN INTERNA DE CARACTERES Y VALORES ESPECIALES	87
CARACTERES.....	87
VALORES ESPECIALES	88
ENDIANNESS.....	89
BIG ENDIAN y LITTLE ENDIAN.....	89
VERIFICANDO CON <code>gdb</code>	91
PROBLEMAS RESUELTOS	94
PROBLEMAS PROPUESTOS	99
4. PUNTEROS A DATOS ATÓMICOS	101
DECLARACIÓN Y ASIGNACIÓN DE PUNTEROS.....	101
TAMAÑO DE UN PUNTERO	102
PUNTEROS EN MEMORIA.....	102
DESREFERENCIA DE UN PUNTERO	103
ARITMÉTICA DE PUNTEROS.....	104
PUNTEROS Y ARREGLOS	106
OPERADORES UNARIOS DE INCREMENTO Y DECREMENTO	106
DECAIMIENTO DE UN ARREGLO.....	106
SUBÍNDICES Y DESPLAZAMIENTOS.....	107
PUNTEROS GENÉRICOS (<code>void*</code>)	108
CONVERSIÓN DE TIPOS DE PUNTEROS (<i>casting</i>)	109
ARREGLO DE PUNTEROS	109
PUNTEROS A PUNTEROS	111

PUNTEROS CONSTANTES Y PUNTEROS A DATA CONSTANTE	112
CADENAS Y PUNTEROS A CARÁCTER.....	113
AGRUPAMIENTO DE CADENAS (<i>string pooling</i>)	114
PROBLEMAS RESUELTOS	115
PROBLEMAS PROPUESTOS	126
5. REPRESENTACIÓN INTERNA DE DATOS AGREGADOS	129
REPRESENTACIÓN INTERNA DE ARREGLOS DE VARIABLES	130
ARREGLOS	130
MATRICES.....	131
CADENAS	131
REPRESENTACIÓN INTERNA DE TIPOS DEFINIDOS POR USUARIO	133
ESTRUCTURAS	133
CAMPOS DE BITS	135
ENUMERACIONES.....	136
UNIONES.....	136
PROBLEMAS RESUELTOS	138
PROBLEMAS PROPUESTOS	143
6. PUNTEROS A DATOS AGREGADOS.....	145
PUNTEROS A ARREGLOS.....	145
ARREGLOS DE PUNTEROS A ARREGLOS	146
PUNTEROS A MATRICES	148
ARREGLOS DE PUNTEROS A MATRICES.....	149
PUNTEROS A ESTRUCTURAS.....	149
PUNTEROS A ARREGLOS DE ESTRUCTURAS	150
COPIA PROFUNDA Y COPIA SUPERFICIAL DE ESTRUCTURAS	150
PROBLEMAS RESUELTOS	152
PROBLEMAS PROPUESTOS	155
7. FUNCIONES.....	157
DEFINICIÓN DE UNA FUNCIÓN.....	158
INVOCACIÓN DE UNA FUNCIÓN.....	158

PASO POR VALOR Y PASO POR REFERENCIA	160
PASO DE ARREGLOS A FUNCIONES	161
PASO DE MATRICES A FUNCIONES.....	162
FUNCIONES QUE RETORNAN PUNTEROS.....	163
PROTOTIPO DE FUNCIONES	164
PROGRAMAS MULTIARCHIVO.....	165
FUNCIONES ESTÁTICAS.....	166
LA FUNCIÓN PRINCIPAL, <code>main</code>	167
RECURSIVIDAD.....	169
DIVIDE Y VENCERÁS.....	169
FUNCIONES RECURSIVAS.....	170
PROBLEMAS RESUELTOS	173
PROBLEMAS PROPUESTOS	194
8. MAPA DE MEMORIA.....	197
SEGMENTOS DE MEMORIA	198
MONTÍCULO (<i>heap</i>).....	198
PILA (<i>stack</i>).....	199
BSS	199
SEGMENTO DE DATOS DE SÓLO LECTURA (<i>rodata</i>).....	200
SEGMENTO DE DATOS INICIALIZADOS.....	200
SEGMENTO DE TEXTO	200
MODIFICADORES	201
VARIABLES AUTOMÁTICAS.....	201
VARIABLES GLOBALES	202
VARIABLES ESTÁTICAS.....	203
CONSTANTES	205
CADENAS	205
9. PUNTEROS A FUNCIONES.....	207
TAMAÑO DE UNA FUNCIÓN.....	209
ARITMÉTICA DE PUNTEROS A FUNCIÓN	209
RETROLLAMADAS (<i>callbacks</i>).....	209

ARREGLO DE PUNTEROS A FUNCIONES	210
CONVERSIÓN DE TIPOS ENTRE FUNCIONES.....	211
RETORNO DE UN PUNTERO A FUNCIÓN	212
PROBLEMAS RESUELTOS	214
PROBLEMAS PROPUESTOS	221
10. ASIGNACIÓN DINÁMICA DE MEMORIA.....	223
SOLICITAR UN BLOQUE DE MEMORIA	224
GRABANDO EN EL MONTÍCULO	225
LIBERACIÓN DE MEMORIA	225
VERIFICANDO LA DISPONIBILIDAD DE MEMORIA.....	226
FUGA DE MEMORIA	226
GRABANDO DATOS DE DIFERENTES TIPOS	226
GRABANDO ESTRUCTURAS DE DATOS.....	227
SOLICITANDO MÁS MEMORIA	228
OTRAS FORMAS DE SOLICITAR MEMORIA	230
PROBLEMAS RESUELTOS	231
PROBLEMAS PROPUESTOS	237
11. ESTRUCTURAS DINÁMICAS DE DATOS.....	239
PILA.....	240
INSERCIÓN DE UN NODO EN UNA PILA.....	240
EXTRACCIÓN DE UN NODO DE UNA PILA.....	241
UNA PILA GENERALIZADA	243
COLA	245
INSERCIÓN DE UN NODO EN UNA COLA.....	245
EXTRACCIÓN DE UN NODO DE UNA COLA.....	246
UNA COLA GENERALIZADA.....	247
ÁRBOL BINARIO	248
ÁRBOL BINARIO DE BÚSQUEDA	248
PROBLEMAS RESUELTOS	251
PROBLEMAS PROPUESTOS	260

CAPÍTULO

1

DATOS ATÓMICOS y DATOS AGREGADOS

Un programa es un conjunto de instrucciones ejecutables que manipulan datos para realizar una determinada tarea. Los datos pueden clasificarse en dos grandes grupos: Los **datos atómicos** son indivisibles y no pueden ser descompuestos en partes significativas más pequeñas, p.ej. el peso de una persona, la edad de un empleado, el número al que deseamos calcular la raíz cuadrada. Por otro lado, los **datos agregados** son agrupaciones de datos relacionados, p.ej. las notas de los estudiantes de un curso, la cantidad de contagios diarios de COVID-19 del mes anterior, la información bancaria de un pago por internet (que suele incluir un número de tarjeta, el nombre del titular, un código secreto, entre otros datos más).

La forma más simple de almacenar un dato atómico consiste en definir una variable de algún tipo predefinido y asignar dicho dato a dicha variable. En cambio, para almacenar datos agregados se debe considerar dos posibles situaciones: (i) cuando todas las componentes del dato agregado son del mismo tipo, p.ej. las notas de todos los alumnos de un curso, o (ii) cuando las componentes del dato agregado son de tipos diferentes, p.ej. la información bancaria de un pago por internet. En el primer caso, el dato agregado podría ser almacenado en un arreglo de variables. En el segundo caso, mucho más general, no existen tipos predefinidos que puedan ayudarnos. El lenguaje C no pudo prever un tipo específico para cualquier información que uno pudiera querer procesar. Cuando se trabaja con esta última clase de datos agregados se debe definir un nuevo tipo de dato, personalizado, para recién, a partir de ese momento, poder declarar variables del tipo que se acaba de definir.

El resto del capítulo está dividido en tres partes donde se estudiará:

- (1) El uso de variables para almacenar datos atómicos,
- (2) El uso arreglos de variables para almacenar datos agregados cuyas componentes son del mismo tipo, y
- (3) El uso de tipos definidos por el programador para almacenar datos agregados de componentes heterogéneas.

VARIABLES

Una **variable** es un nombre que se utiliza para hacer referencia a un grupo de celdas de memoria donde ha de almacenarse algún dato. Para declarar una variable debe especificarse su nombre y su tipo.

El **nombre de una variable** es cualquier secuencia de caracteres que puede incluir letras, dígitos o guiones, _, pero no debe comenzar con dígito. El **tipo de una variable** nos indica las propiedades del dato que esta puede almacenar: el espacio que ocupará en memoria, la forma en que será codificado, las operaciones que pueden realizarse con este y su rango de valores. Para declarar una variable puede usarse cualquiera de los siguientes tipos predefinidos:

Especificador de tipo	Espacio en memoria (en bytes)	Rango
char	1	-128 a 127
unsigned char	1	0 a 255
short	2	-32,768 a 32,767
unsigned short	2	0 a 65,535
int	4	-2,147,483,648 a 2,147,483,647
unsigned int	4	0 a 4,294,967,295
long	8	-9,223,372,036,854,775,808 a +9,223,372,036,854,775,807
unsigned long	8	0 a 18,446,744,073,709,551,615
long long	8	-9,223,372,036,854,775,808 a +9,223,372,036,854,775,807
unsigned long long	8	0 a 18,446,744,073,709,551,615
float	4	3.4E-38 a 3.4E+38
double	8	1.7E-308 a 1.7E+308
long double	16	2.1E-314 a 2.1E+314

Tabla 1.1. Tipos de datos predefinidos

Algunos de estos valores podrían variar dependiendo de la arquitectura con que se trabaje.

DECLARACIÓN DE VARIABLES

La siguiente sintaxis nos permite declarar tres variables:

```
char genero;  
int edad, peso;
```

Declaración de variables

El código anterior contiene dos sentencias. En la primera se ha declarado la variable `genero`; en la segunda se han declarado dos variables: `edad` y `peso`. La variable `genero` es de tipo carácter, `char`; las otras dos, `edad` y `peso`, son de tipo entero, `int`. Luego de sus declaraciones, estas variables ya podrían ser utilizadas para almacenar datos atómicos.

También se pudo haber usado una sentencia propia para cada una de las dos variables enteras, pero, para acostumbrar rápidamente al lector a la riqueza del lenguaje C, se ha preferido hacerlo así, declarar ambas en una misma línea.

ASIGNACIÓN DE VARIABLES

La sintaxis requerida para asignar datos (o valores) a las variables previamente definidas es como sigue:

```
genero = 'M';
edad = 34;
peso = 81;
```

Asignación de variables

En C, el símbolo `=` es utilizado como operador de asignación. Para comparar matemáticamente datos numéricos se utiliza otro símbolo, `==`.

Las instrucciones anteriores podrían leerse como: «Coloca el dato `M` en la variable `genero`; coloca el dato `34` en la variable `edad`; coloca el dato `81` en la variable `peso`». En tiempo de ejecución el sistema procesa las instrucciones de un programa secuencialmente, siempre de arriba abajo.

Las variables de tipo `char` por lo general se utilizan para guardar un carácter (letra, dígito o símbolo alfanumérico) como dato. El carácter a guardar debe estar delimitado entre comillas simples. En el código anterior, también era posible utilizar la asignación `genero = 77` en lugar de `genero = 'M'`. Ambas instrucciones son equivalentes dado que el código ASCII del carácter '`M`' es 77. Cada vez que asignamos un carácter, p.ej. '`@`', a una variable de tipo `char`, estamos asignándole indirectamente un valor entero, el valor que corresponde al código ASCII del carácter asignado, p.ej. 64. En este sentido podríamos decir que las variables de tipo `char` tienen una naturaleza dual: pueden recibir caracteres o números enteros pequeños, según el rango especificado en la Tabla 1.1.

Las variables enteras, `int`, pueden almacenar cualquier valor dentro del rango mostrado en la Tabla 1.1. En nuestro caso, las variables `edad` y `peso` por ahora contienen los datos `34` y `81`, respectivamente. Digo por ahora ya que el dato contenido en una variable podría cambiar muchas veces a lo largo del programa. De allí viene justamente el nombre de «variable». Si se añadiera una cuarta línea `edad=41` al código anterior, el programa seguiría siendo correcto, sólo que en esta situación hipotética el dato inicial, `34`, sería luego actualizado a `41`.

Si uno adicionalmente quisiera guardar un número con parte decimal, p.ej. la estatura de una persona, tendría que utilizar una variable de tipo `float`, `double` o `long double`. Cualquiera de estas variables puede almacenar números decimales. Por conveniencia uno preferiría utilizar el tipo `float` para guardar la estatura. El rango de este tipo es suficiente para almacenar dicho dato. Asignar otros tipos como `double` o `long double` sería menos eficiente ya que, durante la ejecución de nuestro programa, estaríamos utilizando más espacio de memoria del que realmente necesitamos.

También es posible asignarle un valor a una variable en el mismo momento de su declaración. El código siguiente, que ahora incluye la variable `estatura`, muestra la sintaxis requerida:

```
char genero = 'M';
int edad = 34, peso = 81;
float estatura = 1.79;
```

Como se observa en el fragmento de código anterior, los datos enteros suelen ser representados en base decimal dentro de un programa. Esta no es la única opción. También es posible escribirlos en el sistema binario, octal o hexadecimal, usando prefijos apropiados.

```
int x = 0x2B;
int y = 053;
short z = 0b101011;
```

Tal como se muestra arriba, las variables `x`, `y` y `z` han sido asignadas con el mismo valor pero representado de distintas formas. El prefijo `0x` indica que el entero que viene a continuación está escrito en base hexadecimal; el prefijo `0` es para representar enteros en base octal; y `0b` para ingresar números enteros en base binaria.

Si imprimiésemos estas tres variables en base decimal veríamos tres veces el mismo valor, 43.

UNA ASIGNACIÓN NO ES UNA ECUACIÓN

No es extraño el desconcierto de algunos al encontrarse por primera vez con una expresión como:

```
x = x + 10;
```

lo que carecería de sentido si se tratase de una ecuación matemática. Sin embargo, en el ambiente de la programación, la expresión anterior es válida y, de hecho, muy utilizada. Significa que el valor de la variable `x` será incrementado en 10 unidades con respecto a su valor actual.

Cada vez que se ejecuta una asignación el sistema resuelve la expresión que se encuentra a la derecha (*r-value*) del operador de asignación, `=`, y dicho resultado es asignado a la variable referida en el lado izquierdo (*l-value*). El término izquierdo de una asignación debe ser siempre el nombre de una variable; nunca puede ser una expresión algebraica, una constante, ni el nombre de un arreglo, matriz o cadena. En cambio, el término derecho sí puede ser cualquier expresión arbitraria. Según esto, las siguientes expresiones no son asignaciones válidas y causarán un error en tiempo de compilación:

```
15 = 3*x;          // incorrecto
45-7 = edad;      // incorrecto
x+10 = y;          // incorrecto
1-5*y = w+z;      // incorrecto
```

El error causado se conoce como error de operando izquierdo, *l-value error*.

ASIGNACIÓN MÚLTIPLE

Es posible asignar varias variables en una misma sentencia usando la siguiente sintaxis:

```
int x, y, z, w;
x = y = z = w = 100;
```

Asignación múltiple de variables

La sentencia `x = y = z = w = 100` se denomina **asignación múltiple**. Implica una secuencia de asignaciones simples, que es evaluada de derecha a izquierda.

En otras palabras, la asignación `x = y = z = w = 100` es equivalente a las siguientes instrucciones (en el orden mostrado):

```
w = 100;
z = w;
y = z;
x = y;
```

MOSTRAR VARIABLES EN PANTALLA

Es posible mostrar el contenido de nuestras variables en pantalla utilizando la función `printf`. Al invocar esta función es necesario que se informe qué es exactamente lo que se desea imprimir. Para ello debemos utilizar **argumentos**.

El primer argumento de la función `printf` se llama **cadena de formato**. Es un texto delimitado por comillas dobles que indica lo que debe imprimirse en pantalla. La cadena de formato puede incluir, además de expresiones en lenguaje humano, símbolos de la forma `%d`, `%f` y, en general, cualquiera de los mostrados en la primera columna de la Tabla 1.2. Estos símbolos se denominan **especificadores de formato**.

Adicionalmente, la función `printf` suele recibir variables como argumentos. Al momento de imprimir un texto en pantalla la función `printf` reemplaza cada especificador de la cadena de formato por el valor de alguna variable o expresión. Por ejemplo, la instrucción:

```
printf("Mi peso es %d Kg.", peso);
```

imprimirá en pantalla: Mi peso es 81 Kg. (sin las comillas), pues eso es lo que se obtendría luego de reemplazar `%d` con el valor de la variable `peso`. Una expresión como:

```
printf("Peso %d Kg. y mido %f m.", peso, estatura);
```

imprimiría: Peso 81 Kg. y mido 1.790000 m., pues ese sería el contenido de la cadena de formato luego de reemplazar `%d` y `%f` por los valores de `peso` y `estatura`, respectivamente. Como se deduce de estos ejemplos, el número de argumentos que necesita la función `printf` para operar correctamente depende del número de especificadores contenidos dentro de la cadena de formato.

El especificador a utilizar depende del tipo del dato que ocupará su lugar en pantalla. El especificador %d indica que en esa parte de la cadena de formato deberá imprimirse un entero; el especificador %f, un número decimal, etc. La Tabla 1.2 contiene información sobre qué especificador deberá usarse para cada tipo de dato que se desee imprimir.

Especificador de formato	Descripción	Tipos soportados
%c	Carácter	(unsigned) char
%d	Número entero	(unsigned) short int long
%e o %E	Número decimal en notación científicas	float double
%f	Número decimal de precisión simple	float
%g o %G	Similar a %e or %E	float double
%hi	Entero corto con signo (short)	short
%hu	Entero corto sin signo (short)	unsigned short
%i	Entero con signo	(unsigned) short int long
%l %ld %li	Entero largo con signo	long
%lf	Número decimal de precisión doble	double
%Lf	Número decimal con 80 bits de precisión	long double
%lu	Número entero sin signo	unsigned int unsigned long
%lli o %lld	Número entero largo con signo	long long
%llu	Número entero largo sin signo	unsigned long long
%o	Número entero en base octal	(unsigned) short (unsigned) int long
%p	Puntero	void *
%s	Cadena de texto	char *
%u	Número entero sin signo	unsigned int unsigned long
%x o %X	Número entero en base hexadecimal	(unsigned) short (unsigned) int long
%%	Imprime el carácter %	

Tabla 1.2. Especificadores de formato

Por defecto, los números decimales se imprimen con seis dígitos decimales. Esto puede cambiarse usando un formato más específico, p.ej. %.2f para imprimir con dos decimales.

Juntando varios de los fragmentos de código que ya hemos comentado podríamos construir nuestro primer programa, tal como se observa en `prog1.c`.

La primera línea, `#include<stdio.h>`, permite incluir el archivo de cabecera `stdio.h` dentro de nuestro código. Este archivo incluye el prototipo de la función `printf`. Su omisión causaría que no podamos invocar a la función `printf` en nuestro programa. Las instrucciones que parten con el símbolo numeral, #, no son sentencias del lenguaje C; son sentencias que están definidas fuera del lenguaje y son manejadas por el preprocesador, antes de que el compilador comience a verificar la correcta sintaxis del programa.

```
#include <stdio.h>

// mi primer programa
int main(void) {
    char genero = 'M';
    int edad = 34, peso = 81;
    float estatura = 1.79;

    printf("Mi género es %c\n", genero);
    printf("Peso %d Kg. y mido %f m.\n", peso, estatura);
    printf("Actualmente tengo %d años", edad);
}

prog1.c
```

La línea `//Mi primer programa` es un **comentario**. Los comentarios son indicaciones que el programador escribe a lo largo del programa para explicar qué hace su código. Los comentarios facilitan la lectura del código y su uso es muy recomendable, principalmente cuando se trabaja con programas extensos. En C, se debe utilizar dos barras invertidas, `//`, para preceder un comentario corto, de una línea; los comentarios de múltiples líneas deben encerrarse entre los delimitadores `/*` y `*/`. Debe evitar incluir accidentalmente instrucciones dentro de los comentarios, pues si esto ocurriera, dichas instrucciones no serían ejecutadas.

Todas las sentencias discutidas anteriormente deben ir encerradas dentro de la función principal, `main`. Allí es donde comienza la ejecución de un programa. Un programa en C puede tener muchas funciones, pero la función llamada `main`, cuya presencia es obligatoria en cualquier programa, siempre será la primera en ser ejecutada.

EJECUCIÓN DE UN PROGRAMA

Para ejecutar el programa anterior primero debemos compilarlo. Para ello se debe escribir la siguiente instrucción en la línea de comandos:

```
gcc prog1.c
```

Note que `prog1.c` es el nombre del archivo fuente que se va a compilar, el archivo que contiene nuestro código. El resultado de la compilación es un archivo ejecutable de nombre `a.out` si trabaja en Linux, ó `a.exe` si está en Windows. Esos son los nombres que obtenemos por defecto. Si quisieramos otro nombre (p.ej. `milprog.out`) para nuestro archivo ejecutable podríamos hacer:

```
gcc prog1.c -o milprog.out
```

Para ejecutar el archivo `a.out` debe escribir en la línea de comandos:

```
./a.out
```

La salida del programa anterior serían las siguientes tres líneas:

```
Mi género es M  
Peso 81 Kg. y mido 1.79 m.  
Actualmente tengo 34 años
```

Es fácil creer que se han impreso tres líneas porque el código incluye tres sentencias `printf`. No es así. La verdadera razón está en el símbolo '`\n`' que se encuentra al final de las cadenas de formato del primer y segundo `printf`. Este símbolo indica que se haga un salto de línea luego de imprimir un texto en pantalla. De haber omitido estos símbolos las tres líneas que ahora vemos separadas se habrían impreso una a continuación de otra. El símbolo '`\n`' es una secuencia de escape.

Una **secuencia de escape** es un símbolo que se usa para referirse a caracteres que no tienen representación gráfica (p.ej. los caracteres de tabulación o retroceso), o a aquellos que ya tienen un significado específico dentro del lenguaje (p.ej. las comillas dobles).

Otras secuencias de escape conocidas se muestran en la siguiente tabla:

Secuencias de escape	Carácter representado
<code>\n</code>	Nueva línea
<code>\b</code>	Carácter de retroceso
<code>\t</code>	Tabulación horizontal
<code>\v</code>	Tabulación vertical
<code>\a</code>	Produce un sonido
<code>\?</code>	Signo de interrogación, ?
<code>\"</code>	Doble comilla
<code>\'</code>	Comilla simple
<code>\\\</code>	Barra invertida \
<code>\%</code>	Carácter %

Tabla 1.3 Secuencias de escape

DEL TECLADO A LA VARIABLE

En nuestro primer programa, los valores de las variables `genero`, `edad`, `peso` y `estatura` fueron asignados directamente en el código (*hard-coded*, en inglés). Otra forma de asignarle un valor a una variable es permitiendo que el usuario, desde su teclado, pueda ingresar este valor.

El siguiente programa solicita al usuario que ingrese su género, peso, edad y talla, desde el teclado, para luego imprimir los valores ingresados:

```
#include <stdio.h>

int main(void) {
    char genero;
    int edad, peso;
    float estatura;

    printf("Ingrese su genero [M/F]: ");
    scanf("%c", &genero);

    printf("Ingrese su edad: ");
    scanf("%d", &edad);

    printf("Ingrese su peso (Kg.): ");
    scanf("%d", &peso);

    printf("Ingrese su talla (m): ");
    scanf("%f", &estatura);

    printf("Mi género es %c\n", genero);
    printf("Peso %d Kg. y mido %f m.\n", peso, estatura);
    printf("Actualmente tengo %d años", edad);
}
```

prog2.c

Este nuevo programa utiliza la función `scanf`, cuyo prototipo se encuentra en `stdio.h`. Esta función permite que, en plena ejecución del programa, el usuario pueda escribir un dato que terminará siendo almacenado en una variable.

El programa anterior empieza mostrando un mensaje al usuario, `Ingrese su genero [M/F]`. Esto es consecuencia de la primera llamada a la función `printf`. Luego, el sistema ejecuta la instrucción `scanf("%c", &genero)`. En ese momento el programa se quedará suspendido, esperando que el usuario ingrese un dato a través de su teclado, supuestamente 'M' o 'F', el carácter que represente su género. El usuario puede tomarse todo el tiempo que necesite; el programa no continuará ejecutándose hasta que se haya ingresado un dato y presionado la tecla ENTER. Recién en ese momento el dato ingresado se almacenará en la variable `genero` y, solo entonces, el sistema continuará con la ejecución de la siguiente instrucción, el segundo `printf`.

El primer argumento pasado a `scanf`, `%c`, indica el tipo de dato que se ingresará desde el teclado; el segundo, `&genero`, indica la dirección de memoria donde habrá de colocarse el dato leído. Más adelante hablaremos extensamente sobre el operador de dirección, `&`. Por ahora bastará con saber que para que el usuario pueda introducir un dato en una variable `var` deberá pasar `&var` como segundo argumento a `scanf`. Adelanto que existen excepciones, p.ej. cuando se trata de leer cadenas de texto; pero aún es temprano para tratar estos temas.

Siguiendo con el análisis de la primera instrucción `scanf`, no podemos negar la posibilidad de que el usuario ingrese un dato distinto de 'M' o 'F'. En dicho escenario lo deseable sería que el programa advierta al usuario de su error y que le vuelva a solicitar que ingrese su género. Este tipo de **validaciones** se implementa utilizando sentencias condicionales y bucles. Por lo pronto asumiremos que estamos frente a un usuario ideal, uno que actúa precavidamente, esforzándose por evitar los potenciales fallos (*bugs*) que el programador podría haber dejado regados a lo largo del programa.

OPERADORES

Existe una amplia gama de símbolos que nos permite realizar operaciones con los datos de nuestras variables. Los **operadores aritméticos** son binarios y siempre retornan un número:

Símbolo	Nombre	Descripción
+	Suma	15+7 retorna 22
-	Resta	15-7 retorna 8
*	Multiplicación	15*7 retorna 105
/	División	15/7 retorna 2, el cociente de dividir 15 entre 7
%	Módulo	15%7 retorna 1, el residuo de dividir 15 entre 7

Los **operadores de comparación** son binarios y se utilizan para definir expresiones que poseen un valor de verdad (verdadero o falso):

Símbolo	Nombre	Descripción
<	Menor que	La expresión $a < b$ es verdadera si a es menor que b ; de lo contrario es falsa
>	Mayor que	La expresión $a > b$ es verdadero si a es mayor que b ; de lo contrario es falsa
\leq	Menor o igual que	La expresión $a \leq b$ es verdadero si a es menor o igual a b ; de lo contrario es falsa
\geq	Mayor o igual que	La expresión $a \geq b$ es verdadero si a es mayor o igual a b ; de lo contrario es falsa
\equiv	Igual que	La expresión $a \equiv b$ es verdadero si a y b son iguales; de lo contrario es falsa
\neq	Diferente de	La expresión $a \neq b$ es verdadero si a y b son diferentes; de lo contrario es falsa

Los **operadores lógicos** se aplican sobre expresiones booleanas y retornan un valor de verdad:

Símbolo	Nombre	Descripción
$\&\&$	AND	La expresión $\text{expr}_1 \&\& \text{expr}_2$ es verdadera si ambas expresiones, expr_1 y expr_2 , son verdaderas; de lo contrario es falso
$\ $	OR	La expresión $\text{expr}_1 \ \text{expr}_2$ es falsa si ambas expresiones, expr_1 y expr_2 , son falsas; de lo contrario es verdadera
!	NOT	La expresión $!\text{expr}_1$ es falsa si expr_1 es verdadera; de lo contrario es verdadera

El **operador de tamaño**, `sizeof`, funciona en tiempo de compilación. Puede ser invocado de dos formas: (1) pasándole el nombre de una variable, p.ej. `sizeof(myVar)`, en cuyo caso retornará el espacio (en bytes) que ocupará `myVar` en memoria. O, (2) pasándole el nombre de un tipo, p.ej. `sizeof(int)`, en cuyo caso se retornará el espacio (en bytes) que ocuparía cualquier variable que sea de tipo `int`.

El **operador de conversión de tipos** permite convertir explícitamente el tipo de un dato. Un caso muy común donde se requiere conversión de tipos es cuando deseamos forzar una división de números reales para evitar la división entera. La asignación `float x = 15/4`, por ejemplo, asignará el valor de 3.0 a la variable `x`; en cambio, con la conversión de tipos (*casting*), la sentencia `float x = (float)15/4;` le asignará el valor 3.75. La diferencia radica en que el lenguaje sobreentiende que la división de dos enteros debe retornar un resultado entero. Esto se altera al cambiar el tipo del numerador con `(float)15`.

Los **operadores de asignación compuestos**, `+=`, `-=`, `*=`, `/=`, `%=`, son simplificaciones de asignaciones más complejas, tal como se muestra en la siguiente tabla de equivalencias:

Expresión	Equivalencia
<code>x += 10;</code>	<code>x = x + 10;</code>
<code>x -= 10;</code>	<code>x = x - 10;</code>
<code>x *= 10;</code>	<code>x = x * 10;</code>
<code>x /= 10;</code>	<code>x = x / 10;</code>
<code>x %= 10;</code>	<code>x = x % 10;</code>

Los **operadores unarios de incremento**, `++`, y **decremento**, `--`, permiten aumentar o reducir en una unidad el valor de una variable. Cuando no se incrusta dentro de una sentencia más compleja, resulta indiferente colocar el operador de incremento antes o después del nombre de la variable a modificar. Por ejemplo, los siguientes fragmentos de código son equivalentes:

<code>int x=10; x = x + 1;</code>	<code>int x=10; x += 1;</code>	<code>int x=10; x++;</code>	<code>int x=10; ++x;</code>
Distintas formas de incrementar en una unidad el valor de una variable			

Cuando se usa una variable incrementada (con `++`) dentro de otras sentencias más complejas, ya no es lo mismo usar `++` como prefijo que como sufijo. Por ejemplo, la instrucción `printf("%d", x++)` hace dos cosas: (1) imprimir el valor de `x`, y (2) incrementar valor de `x` en una unidad. Por otro lado, `printf("%d", ++x)` hace las mismas dos cosas, pero en diferente orden: Primero incrementa el valor de `x` en una unidad; y luego imprime el valor de `x` ya incrementado.

Los siguientes programas, cuyas salidas están comentadas, son ilustrativos:

<code>int x=10; printf("%d", x++); // imprime 10 printf("%d", ++x); // imprime 12</code>	<code>int x=10; printf("%d", ++x); // imprime 11 printf("%d", x++); // imprime 11</code>
Diferencia entre <code>x++ y ++x</code>	

El operador de decremento, `--`, tiene un comportamiento análogo al que acabamos de describir.

El **operador de tipos**, `typeof`, es una extensión del lenguaje C que permite extraer el tipo de una variable y usarlo en la declaración de otra. Dicha funcionalidad se ejecuta en tiempo de compilación. El siguiente programa imprime las variables `x` e `y`, ambas de tipo flotante:

```
float x = 3.14;
typeof(x) y = 1.41;
printf("%f %f", x, y); // imprime 3.14 y 1.41
```

Cuando varios operadores son utilizados en una misma expresión es necesario conocer el orden en que serán evaluados, para así evitar ambigüedades. Dicho orden se especifica en la Tabla 1.4. Los operadores que se encuentran en las filas superiores (de arriba) tienen mayor precedencia, i.e. se evalúan antes, que los operadores de las filas inferiores.

Según la tabla, para evaluar la instrucción `x=3+5*8`, el sistema realizará primero la multiplicación, `*`, luego la suma, `+`, y finalmente la asignación, `=`.

Operador	Descripción
<code>() [] -> .</code>	Operadores de llamada a función, de acceso a arreglo por subíndice y de acceso a atributos
<code>! * & ++ -- sizeof</code>	Operadores de negación, de desreferencia, de dirección, de incremento/decremento, de tamaño
<code>* / %</code>	Operadores aritméticos de multiplicación, división y residuo
<code>+ -</code>	Operadores aritméticos de suma y resta
<code>< > <= >=</code>	Operadores de desigualdad
<code>== !=</code>	Operadores de igualdad
<code>? :</code>	Operador ternario
<code>= += -= *= /= %=</code>	Operadores de asignación
<code>,</code>	Separador de expresiones

Tabla 1.4. Precedencia de operadores

ARREGLOS DE VARIABLES

Las variables de tipos predefinidos no son adecuadas para almacenar datos agregados de manera eficiente. Si deseáramos, por ejemplo, registrar el dato agregado `{ 6, 9, 7, 6, 8 }`, que representa las notas de cinco estudiantes, sería muy ineficiente tener que definir cinco variables de tipo entero, `int`, una para almacenar cada nota. Si así lo hiciéramos estaríamos creando un programa engorroso, difícil de leer y mantener. Obviamente la situación empeoraría si tuviéramos que almacenar las notas de los centenares de estudiantes que asisten a una universidad. En dicho caso, el número de variables que se requerirían aumentaría considerablemente junto con la complejidad del código.

La solución ideal para guardar datos agregados cuyas componentes son del mismo tipo consiste en utilizar arreglos de variables (o simplemente arreglos, como suele decirse).

ARREGLOS

La sintaxis que se requiere para declarar un arreglo de datos enteros es la siguiente:

```
int notas[5];
```

Los corchetes, `[]`, que siguen al identificador `notas` indican que estamos definiendo un arreglo. El valor dentro de los corchetes, 5, indica el tamaño del arreglo.

Es posible asignar un dato agregado a un arreglo en el momento de su declaración, haciendo:

```
int notas[5] = { 6, 9, 7, 6, 8 };
```

La lista de datos a la derecha del operador de asignación, `=`, es justamente el dato agregado que deseamos guardar en el arreglo `notas`. Si el dato agregado tuviese más elementos que el tamaño del arreglo, el compilador lanzará una advertencia; si tuviese menos, el compilador asumirá que los elementos no declarados son todos ceros. De lo último se deduce que para inicializar en cero cada elemento de `notas` bastaría con declarar `int notas[5] = { 0 }`.

Cuando se asigna un dato agregado a un arreglo al momento de la declaración, tal como en la sentencia anterior, es posible omitir el tamaño del arreglo. O sea, también era válido:

```
int notas[] = { 6, 9, 7, 6, 8 };
```

En este caso el compilador fija el tamaño del arreglo al número de elementos que se está asignando.

La asignación de `{ 6, 9, 7, 6, 8 }` sólo es válida al momento de la declaración de un arreglo. Si luego se quisiera, por ejemplo, aumentar un punto a cada estudiante, la sentencia `notas = { 7, 10, 8, 7, 9 }`

sería rechazada por el compilador. Para llevar a cabo esta tarea tendríamos que actualizar manualmente cada elemento del arreglo `notas`, uno por uno, con las siguientes instrucciones:

```
notas[0] = 7;  
notas[1] = 10;  
notas[2] = 8;  
notas[3] = 7;  
notas[4] = 9;
```

Este último fragmento nos revela claramente la naturaleza de un arreglo. Bajo un mismo nombre tenemos un catálogo de cinco variables enteras, `notas[0]`, `notas[1]...`, `notas[4]`. Esta opción es preferible a tener que definir variables independientes, una para cada nota, p.ej. `int notaPepito;` `int notaLuchito, etc.`

Todas las variables de un arreglo tienen el mismo nombre y sólo se diferencian por un subíndice. Para referirnos a la primera variable del arreglo `notas` (la mayoría prefiere decir el primer elemento de `notas`) debemos utilizar el subíndice **cero**, `notas[0]`. Para sumar, por ejemplo, las notas del segundo y cuarto estudiante, podemos utilizar `notas[1]+notas[3]`, y, en general, para referirnos al *i*-ésimo elemento del arreglo debemos usar `notas[i-1]`.

Debe aclararse que la omisión del tamaño del arreglo sólo está permitida cuando este es asignado al momento de la declaración. De lo contrario, es obligatorio especificar el tamaño de un arreglo.

Lamentablemente no existe un especificador de formato que permita utilizar la función `printf` para imprimir todo un arreglo a partir de su nombre. Para mostrar un arreglo en pantalla se deben imprimir sus elementos uno por uno.

La instrucción `sizeof(notas)` devolverá 20, el tamaño total de las cinco variables enteras que constituyen el arreglo `notas`. Una fórmula muy utilizada para obtener el número de elementos de un arreglo es la siguiente:

```
sizeof(notas)/sizeof(notas[0])
```

Lo que se está haciendo es dividir el tamaño total del arreglo entre el tamaño de su primer elemento. Como todos los elementos de un arreglo son del mismo tipo y, por lo tanto, tienen el mismo tamaño, la fórmula anterior devuelve el número de elementos del arreglo `notas`.

ARREGLOS DE TAMAÑO VARIABLE (VLA)

Un arreglo de tamaño variable (VLA, por sus siglas en inglés) es aquel cuyo tamaño está definido no por un constante, sino por una variable, p.ej. `int arr[N]`, o, en general, por una expresión algebraica, p.ej. `int arr[2*N+5]`. El tamaño de un VLA recién se conoce en tiempo de ejecución.

Es fácil confundirse y creer que un VLA puede cambiar de tamaño durante la ejecución del programa, que primero bien pudiera tener 5 elementos, luego 100, y finalmente 20, por ejemplo. Esto no es así. El tamaño de un arreglo nunca varía. En una declaración de la forma `int arr[2*N+5]`, primero se evalúa el valor de `2*N+5` y, en ese momento, se fijará el tamaño del arreglo `arr`. Este arreglo se quedará de tamaño fijo hasta el final del programa, indiferente a los múltiples cambios de valor que podría experimentar la variable `N`.

El uso de VLAs se presenta naturalmente cuando sólo el usuario final conoce la cantidad de datos que el programa necesita procesar. En dichas situaciones, el usuario, desde el teclado, podría definir el tamaño de un VLA, tal como se aprecia en el siguiente fragmento de código:

```
int num_alumnos;
printf("Ingrese el número de alumnos: ");
scanf("%d", &num_alumnos)
int notas[num_alumnos];           // VLA
```

Es importante conocer algunas restricciones que existen respecto a los VLAs. Primero, los VLAs no pueden ser inicializados, i.e. no se les puede asignar una lista valores, como `{ 5, 8, 2,... }`, por ejemplo, al momento de su declaración. Segundo, la expresión que define el tamaño de un VLA debe ser reducible a un valor entero. No es posible una declaración como `int arr[N]`, si `N` ha sido previamente declarado de tipo flotante, `float`.

MATRICES

Una matriz es un arreglo de arreglos. Conviene concebirlas como una malla bidimensional de celdas, tal que en cada celda es posible almacenar un dato. Aparecen naturalmente cuando se necesita almacenar valores que dependen de dos variables, que bien podrían asociarse una a las filas y otra a las columnas. Por ejemplo, las ventas diarias de tres sucursales durante los últimos 5 días laborables podría representarse así:

	LUN	MAR	MIE	JUE	VIE
Sucursal A	59.6	80.7	20.6	43.2	92.1
Sucursal B	43.2	21.3	21.3	12.2	43.7
Sucursal C	65.3	75.2	22.1	72.3	38.7

El dato agregado mostrado arriba cabe naturalmente en una matriz de números decimales de 3 filas y 5 columnas, que podría ser declarada de la siguiente manera:

```
float Ventas[3][5];
```

El tipo `float` de la matriz `Ventas` se debe a que el dato a almacenar en cada celda debe ser un número decimal. Los dos corchetes, `[] []`, que siguen al identificador `Ventas` indican que estamos definiendo un arreglo de arreglos o matriz. Los valores 3 y 5 indican las dimensiones de la matriz. El primero define el número de filas; el segundo, el de columnas.

Uno podría asignar manualmente un valor a cada celda de la matriz, una por una, escribiendo el siguiente código:

```
Ventas[0][0] = 59.6; Ventas[0][1] = 80.7; ... Ventas[0][4] = 92.1;  
Ventas[1][0] = 43.2; Ventas[1][1] = 21.3; ... Ventas[1][4] = 43.7;  
Ventas[2][0] = 65.3; Ventas[2][1] = 75.2; ... Ventas[2][4] = 38.7;
```

Los puntos suspensivos no son parte de la sintaxis de C. Los he utilizado para no tener que escribir todas las asignaciones requeridas, pues estoy seguro que con las asignaciones mostradas el lector podrá deducir las faltantes.

Para referirnos a un elemento particular de la matriz se necesita utilizar dos subíndices. El primero indica la fila del elemento al que deseamos referirnos; el segundo, la columna del mismo. Al igual que con los arreglos los índices de una matriz empiezan desde cero, p.ej. el tercer elemento de la primera fila sería `Ventas[0][2]` y, en general, el j -ésimo elemento de la i -ésima fila sería `Ventas[i-1][j-1]`.

También era posible asignar los valores de `Ventas` al momento de su declaración:

```
float Ventas[3][5] = { {59.6, 80.7, 20.6, 43.2, 92.1},  
                      {43.2, 21.3, 21.3, 12.2, 43.7},  
                      {65.3, 75.2, 22.1, 72.3, 38.7} };
```

Las llaves anidadas, `{ {...}, {...}, {...} }`, utilizadas en la inicialización anterior vuelven a recordarnos que estamos frente a un arreglo de arreglos.

Cuando se inicializa una matriz al momento de su declaración está permitido omitir su número de filas, pues esto puede darse del dato agregado asignado. O sea, también era válido hacer:

```
float Ventas[] [5] = { {59.6, 80.7, 20.6, 43.2, 92.1},  
                      {43.2, 21.3, 21.3, 12.2, 43.7},  
                      {65.3, 75.2, 22.1, 72.3, 38.7} };
```

Bajo ninguna circunstancia puede omitirse el número de columnas de una matriz.

La expresión `sizeof(Ventas)` retorna 60, el espacio ocupado por las 15 variables de tipo `float` que subyacen bajo el nombre `Ventas`. La expresión `sizeof(Ventas[0])` retorna 20, el tamaño total de todos los elementos de la primera fila. Sabiendo esto, las dimensiones de una matriz pueden obtenerse con las siguientes fórmulas:

```
int num_filas = sizeof(Ventas) / sizeof(Ventas[0]);
int num_columnas = sizeof(Ventas[0]) / sizeof(Ventas[0][0]);
```

Como todos los elementos de una matriz tienen el mismo tipo y, por ende, el mismo tamaño, las filas también tendrán el mismo tamaño. Por esto, las fórmulas anteriores seguirían siendo válidas incluso si en lugar de utilizar `Ventas[0]` se usara `Ventas[1]` o `Ventas[2]`, y `Ventas[1][3]` o cualquier otro elemento en lugar de `Ventas[0][0]`.

Finalmente, debe decirse que también es posible inicializar una matriz usando un único par de llaves.

```
float Ventas[3][5] = { 59.6, 80.7, 20.6, 43.2, 92.1,
                      43.2, 21.3, 21.3, 12.2, 43.7,
                      65.3, 75.2, 22.1, 72.3, 38.7};
```

En este caso, los elementos de la matriz serán asignados por filas (primero los elementos de la primera fila, luego los de la segunda, etc.), tomando los datos de la lista inicializadora, uno por uno, en orden de aparición. Si la lista inicializadora no tuviera los quince, 3×5 , datos requeridos para cubrir todas las celdas de la matriz, las últimas posiciones serán asignadas con ceros. De lo último se deduce que para declarar una matriz llena de ceros bastaría con hacer `float Ventas[3][5] = {0};`

CADENAS

Existe un tipo especial de arreglo llamado cadena. Una cadena es un arreglo de caracteres que termina en un símbolo especial denominado carácter de terminación, denotado por '`\0`'.

Las cadenas sirven para guardar textos de longitud arbitraria. Por ejemplo, para guardar el nombre de una persona se podría utilizar la siguiente sentencia:

```
char nombre[10] = {'P', 'e', 'd', 'r', 'o', '\0'};
```

que sería equivalente a esta otra, mucho más sucinta:

```
char nombre[10] = "Pedro";
```

Cualquiera de las dos declaraciones anteriores también equivale a lo siguiente:

```
char nombre[10];
nombre[0] = 'P'; nombre[1] = 'e'; nombre[2] = 'd';
nombre[3] = 'r'; nombre[4] = 'o'; nombre[5] = '\0';
```

El carácter de terminación '\0' (una barra invertida seguida de cero) es utilizado por varias funciones (p.ej. `printf`) para determinar dónde termina una cadena. Por ejemplo, la instrucción:

```
printf("%s", nombre);
```

imprimirá solamente Pedro. La función `printf` sabe cuándo detenerse. No imprimirá los caracteres almacenados en `nombre[6]`, `nombre[7]`, ..., `nombre[9]`, que podrían estar conteniendo cualquier dato. Sólo imprimirá los primeros cinco caracteres y dejará de seguir imprimiendo apenas encuentra el carácter de terminación, '\0'.

Si el texto asignado a una cadena fuera más largo que el tamaño con el que esta ha sido definida, el compilador lanzará una advertencia: «el texto usado en la inicialización es demasiado largo».

Para asignar un texto a una cadena debemos considerar que siempre necesitaremos un carácter adicional para guardar el carácter de terminación '\0'. Así, el arreglo de tamaño mínimo que se requeriría para almacenar "Pedro" sería de 6 caracteres, i.e. `char nombre[6]`.

Debe tenerse cuidado cuando se intente cambiar el valor de `nombre` después de su declaración. No estaría permitido hacer algo como:

```
char nombre[10] = "Pedro";
nombre = "Evo";
```

La última línea arrojaría un error. Como ya se dijo anteriormente, no es posible utilizar el nombre de una cadena en el lado izquierdo de una asignación. Y eso justamente lo que se ha hecho en `nombre = "Evo"`. Alguien podría sentirse confundido al recordar que hace unas pocas líneas atrás hemos utilizado expresiones como `nombre[0] = 'P'`. Pero, en este caso, lo que está a la izquierda del operador de asignación, `nombre[0]`, es el nombre de una variable, no de un arreglo, matriz o cadena.

La manera correcta de proceder con la actualización deseada sería así:

```
nombre[0] = 'E';
nombre[1] = 'v';
nombre[2] = 'o';
nombre[3] = '\0';
```

Otra forma más directa sería haciendo `strcpy(nombre, "Evo")`. Esto último requeriría incluir el archivo de cabecera `string.h` en nuestro programa.

La expresión `sizeof(nombre)` retornaría el valor de 10, el tamaño con que se definió la cadena `nombre`. Muchas veces es importante saber el tamaño del texto; es decir, de los caracteres significativos. Para esto suele usarse `strlen(nombre)` que, en nuestro caso, luego de la actualización

de `nombre`, arrojaría 3, la cantidad de caracteres contenidos en `Evo`. Note que `strlen` no cuenta el carácter de terminación. El prototipo de `strlen` también se encuentra en el archivo `string.h`.

Las cadenas no necesariamente deben ser almacenadas en arreglos de caracteres. Otra forma de almacenar un texto es a través de la siguiente sintaxis:

```
char* nombre = "Pedro";
```

La última sentencia utiliza un puntero a caracteres, `char*`. El manejo de `nombre` es distinto cuando se define como puntero a carácter que cuando se define como arreglo de caracteres. Por ejemplo, ahora sí sería válido algo como:

```
nombre = "Evo";
```

Al compilador no le molesta la asignación anterior puesto que lo que está escrito a la izquierda de la misma no es un nombre de arreglo, matriz o cadena, sino el nombre de una simple y sencilla variable (de tipo puntero). Por lo pronto no diremos más al respecto. El tema de punteros se estudiará extensamente más adelante.

IMPRIMIR EN UNA CADENA. USO DE `sprintf`

Así como la función `printf` nos permite escribir un texto en pantalla, la función `sprintf` permite escribir un texto dentro de un arreglo de caracteres. Antes ya dijimos que la instrucción

```
printf("Peso %d Kg. y mido %f m.", peso, estatura);
```

hace que se imprima el mensaje: Peso 81 Kg. y mido 1.790000 m., en la pantalla. Similarmente, la instrucción

```
sprintf(txt, "Peso %d Kg. y mido %f m.", peso, estatura);
```

no generará ningún mensaje en pantalla, pero escribirá: Peso 81 Kg. y mido 1.790000 m. dentro de la variable `txt`, que debe haber sido previamente declarada como cadena, p.ej. `char txt[100]`.

Imprimir en una cadena puede ser útil en varios casos. Por ejemplo, cuando queremos ejecutar, desde nuestro programa, algún comando del sistema operativo que involucre variables asignadas por el usuario. Esto último puede lograrse con apoyo de la función `system` (en `stdlib.h`), tal como muestra el siguiente programa:

<pre>#include <stdio.h> #include <stdlib.h> int main(void) { char comando[100]; int anho, mes; printf("Ingrese mes:"); scanf("%d", &mes); printf("Ingrese anho:"); scanf("%d", &anho); sprintf(comando, "cal %d %d", mes, anho); system(comando); }</pre>	<pre>User@DESKTOP-ETMGBI ~ \$./a Ingrese mes:9 Ingrese anho:1945 September 1945 Su Mo Tu We Th Fr Sa 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30</pre>
---	--

Una posible salida del programa de la izquierda es mostrada al lado derecho de la tabla anterior.

Este programa no dibuja manualmente el calendario, sino que invoca a la función `cal`, un conocido comando bash. La misma salida que se muestra arriba podría haberse obtenido también escribiendo directamente el comando `cal 9 1945` en la línea de comando del sistema operativo.

La idea central detrás del programa anterior consiste en usar `sprintf` para construir cadenas de la forma "`cal 9 1945`", "`cal 11 1980`", "`cal 4 2012`", etc., según el mes y el año que ingrese el usuario en cada ejecución. Dichas cadenas son luego pasadas como parámetro a la función `system` para que el sistema operativo, a petición de nuestro programa, pueda imprimir el calendario de septiembre de 1945, noviembre de 1980 o abril del 2012, respectivamente.

Los datos que ingresa el usuario se almacenan en las variables `mes` y `anho`. Como se observa en la llamada a `sprintf`, la cadena `comando` tiene el formato "`cal %d %d`", donde los especificadores `%d` y `%d` son reemplazados por los valores de `mes` y `anho` previamente ingresados por el usuario.

La última línea del programa, `system(comando)`, equivale a escribir el contenido de `comando` directamente en el bash.

VARIABLES DE TIPOS DEFINIDOS POR USUARIO

Los tipos predefinidos (`short`, `int`, `double`, `char`, etc.) fueron suficientes para almacenar datos atómicos; los arreglos de tipos predefinidos bastaron para almacenar datos agregados cuyos componentes eran del mismo tipo; pero para guardar datos agregados de componentes heterogéneas, como, por ejemplo, la información civil de un ciudadano, `{10092105, "Juan Pérez", 15-10-1980, "soltero"}`, o la información geográfica de una ciudad, `{"Lima", "Perú", -12.046374, -77.042793}`, tendremos que definir nuestros propios tipos de datos.

El lenguaje C provee la sintaxis necesaria para que el programador pueda definir tipos de datos personalizados, que puedan servir luego como plantillas para almacenar datos agregados de componentes heterogéneas. Estos tipos personalizados se llaman **tipos de datos definidos por usuario**, entendiéndose al programador como el usuario del lenguaje.

ESTRUCTURAS DE DATOS

La siguiente sintaxis permite definir un tipo de dato personalizado:

```
struct empleado {
    char *nombre;
    int edad;
    float salario;
};

int main(void) {
    // código del programa
}

Definición del tipo struct empleado
```

El código anterior añade un nuevo tipo al repertorio de tipos predefinidos que teníamos a nuestra disposición (`short`, `int`, `double`, etc.). El nuevo tipo se llama `struct empleado`. Muchas de las cosas que solíamos hacer con los tipos predefinidos también pueden hacerse con el nuevo tipo: Podemos declarar variables de tipo `struct empleado`, asignar valores a dichas variables, y declarar arreglos y matrices de tipo `struct empleado`, por citar algunas posibilidades.

El tipo `struct empleado` es una estructura. Las estructuras no son los únicos tipos que el programador puede definir; también existen las uniones, campos de bits y las enumeraciones.

El tipo `struct empleado` ha sido definido para almacenar tres datos: una cadena que representa el nombre de una persona, un entero que representa su edad y un número real que representa su salario. Datos agregados como `{"Juan", 23, 3370.8}` pueden ser convenientemente almacenados en una variable de este tipo.

Las variables `nombre`, `edad` y `salario` se denominan elementos de la estructura o **atributos**.

El siguiente código muestra distintas maneras de declarar y asignar variables de tipo `struct empleado`:

```
int main(void) {
    struct empleado e1 = {"Juan", 23, 3370.8};

    struct empleado e2;
    e2.nombre = "Luis";
    e2.edad = 27;
    e2.salario = 4500.9;

    struct empleado e3 = {"Pepe", 34};
    e3.salario = 5081.7;

    struct empleado e4;
    e4 = (struct empleado) {"Ana", 24, 3910.2};
}
```

Distintas maneras de declarar y asignar datos a una estructura

Las variables `e1`, `e2`, `e3` y `e4` son estructuras del tipo `struct empleado`. La variable `e1` es declarada y asignada en una misma línea. La variable `e2` es declarada para posteriormente asignarle valor a cada uno de sus atributos, uno por uno. Note que para acceder a un atributo se utiliza el operador punto entre el nombre de la variable y el nombre del atributo. Para sumar los salarios del primer y segundo empleado, por ejemplo, se debe usar `e1.salario + e2.salario`. En cuanto a la variable `e3`, dos de sus atributos son asignados inmediatamente, al momento de la declaración, y, posteriormente, en la línea siguiente, se le asigna el tercer atributo. Finalmente, la variable `e4` es declarada para luego, en una sentencia posterior, asignar todos sus atributos en una sola línea.

Es posible realizar una asignación como `struct empleado e1 = {0}`. Esto haría que todos los bits ocupados por la variable `e1` se anulen. En este caso, los valores `e1.edad` y `e1.salario` se harían ceros, y el puntero `nombre` tendría valor nulo.

No existe un especificador de formato que permita referirse a toda la estructura. La impresión de una estructura exige la impresión individual de cada uno de sus atributos.

El tipo `struct empleado` será reconocible en todo el programa siempre que haya sido declarado en el ámbito global, i.e. fuera de todas las funciones. Si se hubiese declarado dentro de una función específica, sólo sería reconocible en esa función pero no en otras.

Se puede definir un alias asociado al tipo `struct empleado` escribiendo la siguiente sentencia debajo de su definición:

```
typedef struct empleado Emp;
```

Lo anterior nos permitiría utilizar indistintamente el alias `Emp` o `struct empleado` durante el resto del programa para referirnos al tipo que acabamos de definir. También era posible declarar el alias en la misma sentencia que se utilizó para declarar el tipo `struct empleado`, haciendo:

```

typedef struct empleado {
    char *nombre;
    int edad;
    float salario;
} Emp;

int main(void) {
}

```

Definición del tipo `struct empleado` con alias `Emp`

El código siguiente muestra cómo podría definirse y manipularse un arreglo de empleados:

```

#include <stdio.h>

typedef struct empleado {
    char *nombre;
    int edad;
    float salario;
} Emp;

int main(void) {
    Emp emps[4]; // arreglo de empleados
    emps[0]=(Emp){"Juan", 23, 3370.8};
    emps[1]=(Emp){"Luis", 27, 4500.9};
    emps[2]=(Emp){"Juan", 23, 3370.8};
    emps[3]=(Emp){"Ana", 24, 3910.2};

    float total;
    total = emps[0].salario + emps[1].salario
        + emps[2].salario + emps[3].salario;

    printf("El salario total es %f", total);
}

```

Definición del tipo `struct empleado` con alias `Emp`

Este programa almacena cuatro datos agregados dentro del arreglo `emps` e imprime el salario total de los cuatro empleados, el cual es previamente almacenado en la variable `total`.

Dentro de la función principal, `main`, se pudo haber escrito `struct empleado` en lugar de `Emp` y nada habría cambiado, excepto el tamaño del código fuente.

Es posible declarar varias estructuras dentro un mismo programa según las necesidades del programador. También es posible anidar estructuras; es decir, definir una estructura que posea otra estructura como atributo. Un ejemplo de esto último se observa a continuación:

```

#include <stdio.h>

typedef struct coordenadas {
    double latitud;
    double longitud;
} Coord;

```

```

typedef struct ciudad {
    char* nombre;
    char* pais;
    Coord ubicacion;
} Ciudad;

int main(void) {
    Ciudad c1;
    c1.nombre = "Lima";
    c1.pais = "Peru";
    c1.ubicacion.latitud = -12.04637;
    c1.ubicacion.longitud = -77.042793;

    Ciudad c2 = {"Quito", "Ecuador", {-0.22985, -78.5249}};

    Ciudad c3 = {"Madrid", "España"};
    c3.ubicacion = (Coord) {40.4165, -3.70256};
}

```

Estructuras anidadas

En el código anterior se han definido dos tipos, `struct coordenadas` y `struct ciudad`, con alias `Coord` y `Ciudad`, respectivamente. Los atributos de `struct coordenadas` son dos números reales donde se han de guardar la longitud y latitud de un punto del planeta en coordenadas decimales. Los atributos de `struct ciudad` son: (1) una cadena para almacenar el nombre de una ciudad, (2) una cadena para guardar el nombre del país al que pertenece, y (3) un dato de tipo `struct coordenadas` donde han de guardarse la longitud y latitud de la ciudad.

Para que `struct ciudad` pueda tener un atributo de tipo `struct coordenadas`, este último tipo tiene que haber sido previamente definido, tal como ocurre en el código anterior.

Dentro de la función principal, las variables `c1`, `c2` y `c3` son de tipo `struct ciudad`. Sus valores han sido asignados de distintas formas. En el caso de `c1` se ha accedido individualmente a cada uno de sus atributos. Note que `c1.ubicacion` es de tipo `Coord` y, por lo tanto, tiene dos componentes. Se necesita utilizar el operador punto dos veces para poder acceder al atributo de un atributo, i.e. `c1.ubicacion.latitud` y `c1.ubicacion.longitud`. La definición de `c2` ha sido la más directa. Las llaves anidadas que se observan en la asignación, `{...{...}}`, reflejan que existe una estructura dentro de otra estructura. Finalmente, los datos de la variable `c3` han sido asignados en dos partes: Primero el nombre de la ciudad y del país al que pertenece , y luego el dato agregado `{40.4165, -3.70256}`, de tipo `Coord`, que es almacenado en `c3.ubicacion`.

ENUMERACIONES

Las enumeraciones suelen ser útiles cuando se necesita definir variables que almacenen, por ejemplo, el estado civil, el sexo, el tipo de sangre de una persona, el día de la semana o el mes del año. En todos estos casos, los valores que podrían tomar estas variables son pocos y conocidos de antemano, y podrían ser enumerados en una lista.

La sintaxis para declarar una enumeración es la siguiente:

enum grupo_sanguineo {A, B, AB, O};

Definición del tipo enum grupo_sanguineo
--

Con la sentencia anterior se ha definido un nuevo tipo de datos, una enumeración. Ahora es posible definir variables del tipo enum grupo_sanguineo haciendo:

enum grupo_sanguineo ts;

Declaración de variable del tipo enum grupo_sanguineo

La variable ts puede ser luego asignada de la siguiente forma:

ts = AB;

Asignación de variable del tipo enum grupo_sanguineo
--

Note que no se necesita encerrar AB. Los identificadores, A, B, AB, y O usados en la definición de enum_grupo, se llaman **enumeradores**. Estos pueden utilizarse en cualquier parte del programa, como si fueran constantes enteras. Para imprimir el valor de ts tendríamos que usar el especificador %d. En este caso, el valor que veríamos en pantalla sería 2, porque esta es la posición que ocupa el elemento AB en la lista {A, B, AB, O} considerando que el primer elemento, A, ocupa la posición cero, 0.

Cada vez que definimos una enumeración estamos definiendo una lista de constantes enteras con nombre. Si quisieramos alterar los valores que implícitamente se asignan a los enumeradores podríamos hacer una declaración como la siguiente:

enum grupo_sanguineo {A=20, B=12, AB=57, O=42};

Otra definición del tipo enum grupo_sanguineo

En este último caso la asignación enum grupo_sanguineo ts = AB; terminaría asignando el valor 57 a la variable ts. Otra posible forma de declarar una enumeración sería la siguiente:

enum grupo_sanguineo {A=20, B, AB, O=42};

Otra definición del tipo enum grupo_sanguineo

En este caso los enumeradores que no reciben un valor explícito son añadidos en una unidad con respecto al enumerador anterior. En el último ejemplo, B y AB tomarán los valores 21 y 22, respectivamente.

Algo que no me agrada mucho de las enumeraciones es que uno podría asignar cualquier valor entero a estas variables. Por ejemplo, la sentencia enum grupo_sanguineo ts = 6298 no producirá ninguna advertencia en tiempo de compilación ni errores en tiempo de ejecución. Personalmente hubiera preferido algo como: "Valor fuera de rango para ts", pero no ocurre así. Tampoco me agrada que se pueda utilizar los enumeradores en cualquier lugar del programa. La asignación int edad = AB no

producirá ninguna advertencia o error, y asignará 22 a la variable `edad`, que no es de ningún tipo enumerado. Es cierto que el uso de enumeradores puede hacer nuestros programas más fáciles de leer; pero el programador debe cuidarse de no utilizar enumeradores en cualquier parte del código, pues este podría volverse muy confuso.

Tal como ocurría con las estructuras, también es posible definir alias para las enumeraciones. A continuación se muestra un programa completo:

```
typedef enum grupo_sanguineo {
    A, B, AB, O
} Grupo;

typedef enum estado_civil {
    soltero, casado, viudo, divorciado
} ECivil;

int main(void){
    char nombre[][6] = {"Pedro", "Luis"};
    Grupo tps[2] = {AB, O};
    ECivil ecs[2] = {casado, divorciado};

    printf("%s está %d y su tipo es %d\n", nombre[0], ecs[0], tps[0]);
    printf("%s está %d y su tipo es %d\n", nombre[1], ecs[1], tps[1]);
}
```

Los tipos `enum grupo_sanguineo` y `enum estado_civil` han sido rebautizados con los aliases `Grupo` y `ECivil`, respectivamente. Estos alias se usaron para declarar el arreglo de tipos sanguíneos `tps` y el arreglo de estados civiles `ecs`. El programa arrojará la siguiente salida:

```
Pedro está 1 y su tipo es 2
Luis está 3 y su tipo es 3
```

Usando sentencias condicionales es posible reemplazar los valores numéricos mostrados por sus respectivas etiquetas.

CAMPOS DE BITS

Aunque muchas veces no lo notemos nuestros programas suelen ocupar mucho más espacio en memoria del que realmente necesitan. Cada vez que guardamos la edad de una persona o el nivel académico de un estudiante en una variable entera estamos utilizando 4 bytes (32 bits) para almacenar dichos datos, que bien podrían ser codificados con unos pocos bits. Asumiendo que una persona puede vivir hasta 120 años, su edad podría ser codificada con 7 bits solamente; y, en los países donde las carreras universitarias constan de 10 semestres académicos, este dato puede ser sobradamente almacenado en apenas 4 bits.

Los campos de bits permiten especificar el número de bits que ocuparía un dato entero. En lugar de quedar siempre limitados a enteros de 1 byte (`char`), 2 bytes (`short`), 4 bytes (`int`), y 8 bytes (`long`),

los campos de bits permiten definir enteros de 12 bits, de 5 bits, etc. o incluso de un único bit si así lo deseamos. La sintaxis para definir un campo de bits es la siguiente:

```
struct estudiante {
    char *nombre;
    unsigned int sexo:1;
    unsigned int categoria:2;
    unsigned int semestre:4;
};
```

La estructura `estudiante` tiene 4 atributos. El primero, `nombre`, es normal, digámoslo así; los atributos `sexo`, `categoria` y `semestre` son **campos de bits**, de 1 bit, de 2 bits y de 4 bits, respectivamente. Los campos de bits siempre deben ser definidos dentro de una estructura.

Los tamaños de los campos de bits mostrados son suficientes para codificar el sexo (femenino, masculino), la categoría socioeconómica (A, B, C, D) o el semestre (1-10) de un estudiante. El modificador `unsigned` indica que todos los bits serán utilizados para representar un valor. Si, por ejemplo, se hubiese definido `signed int semestre:4`, sólo se usarían 3 bits para codificar el semestre; el cuarto sería para codificar el signo. En dicho caso, sólo podríamos asignarle valores de 1 a 7 a `semestre` (y también absurdos valores negativos).

Los campos de bits se asignan como si se fueran variables enteras. A continuación se muestra un programa donde se declaran y asignan tres campos de bits:

```
#include <stdio.h>

enum sexo {femenino, masculino};
enum categ_socioecon {A, B, C, D};

typedef struct estudiante {
    char *nombre;
    unsigned int sexo:1;
    unsigned int categoria:2;
    unsigned int semestre:4;
} Alumno;

int main(void) {
    Alumno a1 = {"Ernesto", masculino, B, 10};

    Alumno a2;
    a2.nombre = "Armando";
    a2.sexo = masculino;
    a2.categoria = C;
    a2.semestre = 5;
}
```

No es obligatorio definir enumeraciones para asignar valores a un campo de bits. Por ejemplo, la asignación `a2.categoria = C` bien pudo ser `a2.categoria = 2`. Las enumeraciones sólo se han usado para hacer más legible el código.

En mi computador, tanto `a1` como `a2` ocupan 16 bytes cada uno, a pesar que, en teoría, la estructura `Alumno` podría almacenarse en tan solo 9 bytes: 8 para el atributo `nombre` y apenas un byte (7 bits para

ser más precisos, 1+2+4) para los tres campos de bits. El ahorro de espacio al usar campos de bits no suele ser tan grande como uno esperaría y depende de la arquitectura de la máquina. De cualquier forma, los 16 bytes ocupados por la variable `a2` es preferible a los 20 bytes que habría ocupado si los atributos `sexo`, `categoria` y `semestre` hubiesen sido declarados de tipo entero, `int`.

Los campos de bits no suelen ser muy utilizados en la práctica y hay quienes afirman que la ejecución de programas que utilizan este tipo de datos no es tan eficiente como cuando se trabaja con los tipos de dato estándares.

La representación interna de un campo de bits (la forma en que se almacena dentro de la memoria) no está estandarizada y depende del compilador.

PROBLEMAS RESUELTOS

1. Intercambiar el valor de dos variables

Declare dos variables `x` e `y`, cuyos valores deben ser asignados desde el teclado. Luego, intercambie los valores de dichas variables, i.e. el dato contenido en `x` debe pasar a la variable `y` y viceversa.

Solución:

Este sencillo problema suele tornarse difícil para quienes aún no comprenden bien la operación de asignación. La mayoría suele empezar ensayando instrucciones del siguiente tipo:

```

x = y;
y = x;
```

Lo anterior es incorrecto. Es cierto que si leemos aisladamente las instrucciones anteriores la primera, `x = y`, indica que el dato contenido en `y` se copie a la variable `x`; y la segunda, que el dato de `x` se copie a `y`. Parece ser lo que deseamos, pero no es así. No olvide que las dos instrucciones anteriores no están aisladas sino sujetas a una dependencia temporal, se ejecutan una después de otra. Es fácil olvidar que al ejecutarse `y = x`, el valor de `x` ya ha cambiado en la línea anterior.

Un programa correcto que realice lo solicitado puede escribirse de la siguiente forma:

```
#include <stdio.h>
int main(void) {
    int x, y, tmp;
    printf("Ingrese x:"); scanf("%d", &x);
    printf("Ingrese y:"); scanf("%d", &y);
    tmp = x;
    x = y;
    y = tmp;
    printf("Los valores invertidos son x=%d, y=%d", x, y);
}
```

Se requiere guardar el valor inicial de `x` en una variable temporal `tmp`. Esto es indispensable pues, de lo contrario, la asignación `x = y` haría que se pierda el valor inicial de `x` para el resto del programa.

2. Una forma sencilla de implementar cambios de base

Escriba un programa que solicite al usuario un número en base decimal y le muestre su representación en bases octal y hexadecimal.

Solución:

Un programa correcto sería como sigue:

```
#include <stdio.h>
int main(void) {
    int N;
    printf("Ingrese N:");
    scanf("%d", &N);
    printf("El valor ingresado es %o en base8 y %X en base16", N, N);}
```

Al final, en la última línea, se vuelve a imprimir `N`, el mismo valor ingresado por el usuario, sólo que en distintos formatos: en formato octal, con `%o`, y en hexadecimal, con `%X`.

3. Escriba un programa que solicite al usuario el radio de una esfera y muestre su volumen.

Solución:

Sin necesidad de escribir todo el programa, el error común que suele cometerse en estos problemas consiste en escribir la fórmula del volumen de una esfera como sigue:

```
float volumen = 4/3 * 3.1416 * r * r * r;
```

El sutil error está en la fracción $4/3$. Al ser tanto el numerador como el denominador números enteros, el sistema resuelve $4/3$ como una división entera. No retorna todos los decimales de $4/3$, sino solamente el cociente de dividir 4 entre 3, o sea 1. Para evitar la división entera, por lo menos una de las partes, el numerador o el denominador, debe ser convertida a número decimal. Se podría usar $4.0/3$ ó $4/3.0$ ó $(float)4/3$ en lugar de $4/3$ para corregir la fórmula anterior.

Otra manera correcta de implementar la fórmula también es la siguiente:

```
float volumen = 4 * 3.1416 * r * r * r / 3;
```

El numerador, por ser el resultado de multiplicar enteros y números decimales, es considerado como un número decimal que, al ser luego dividido por el entero 3, arroja un resultado decimal. En general, las operaciones binarias entre dos datos del mismo tipo retornan un resultado de este tipo común.

En cambio, en operaciones binarias entre dos datos de diferentes tipos, el resultado será del tipo que posea mayor rango.

4. Escriba un programa que solicite al usuario una ecuación de primer grado de coeficientes enteros, $ax+b=c$, y muestre la solución de la misma. Algunas salidas podrían ser:

```
User@DESKTOP-ETMGBI ~
$ ./a
Ingrese ecuación:5x+30=50
La solución es 4.000000
User@DESKTOP-ETMGBI ~
$ ./a
Ingrese ecuación:3x+2=12
La solución es 3.333333
User@DESKTOP-ETMGBI ~
$ ./a
Ingrese ecuación:4x+10=-20
La solución es -7.500000
```

Solución:

En las salidas mostradas se observa que el usuario ingresa las ecuaciones con un formato específico: un entero seguido de la variable x , el signo $+$, otro entero, el signo $=$, y, finalmente, un tercer entero que vendría a ser resultado. Por ello, el programa deberá leer desde el teclado un texto cuyo formato sea $\%dx+\%d=\%d$. El programa pedido podría quedar como sigue:

```
#include <stdio.h>
int main(void) {
    int a, b, c;
    printf("Ingrese ecuación:");
    scanf("%dx+\%d=\%d", &a, &b, &c);
    float resultado = (float)(c-b)/a;
    printf("La solución es %f", resultado);
}
```

La ecuación ingresada por el usuario es leída con la función `scanf`. Especificando el formato correcto, $\%dx+\%d=\%d$, es posible identificar cada coeficiente a , b y c de la ecuación.

Matemáticamente, la solución de $ax+b=c$ es $x=(c-b)/a$; pero computacionalmente hay que considerar un detalle adicional. Aunque los coeficientes a , b y c sean enteros, el valor de x podría ser un número decimal. Por ello, sería incorrecto imprimir $(c-b)/a$ directamente. En esta última división, el numerador y el denominador son enteros y, por ende, el resultado también será un entero. Para evitar una posible imprecisión a causa de la división entera, cualquiera de los miembros de la división debe convertirse a flotante.

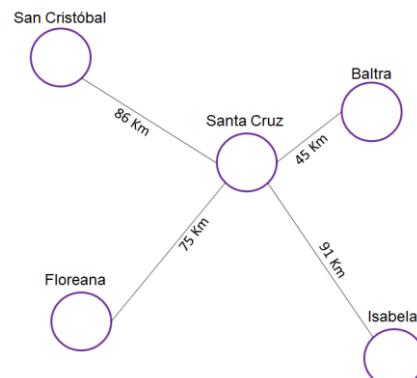
En el programa anterior, la expresión `(float) (c-b) / a` convierte el numerador al tipo flotante para forzar la división exacta. El resultado de esta operación es guardado en la variable `resultado`, desde donde es impreso en la última línea.

5. Tras los pasos de Charles Darwin

Las islas Galápagos constituyen un archipiélago de trece islas ubicadas a 1000 Km. de las costas de Ecuador. Su diversidad de flora y fauna inspiró a Charles Darwin a formular su teoría de la evolución de las especies.

Actualmente, los circuitos turísticos más conocidos suelen partir de la céntrica isla de Santa Cruz hacia otras cuatro islas, tal como se aprecia en el gráfico.

Se le pide implementar un programa que almacene la información mostrada en el gráfico adjunto.



Solución:

Cuando se trabaja con problemas de mapas el primer paso suele ser «cargar» la información del mapa a la memoria para que, desde aquí, por medio de muchos cálculos, se pueda luego determinar, por ejemplo, la ruta más corta entre dos lugares o el circuito más corto que incluya un determinado subconjunto de ciudades. Por ahora debemos contentarnos con representar el mapa mostrado en variables.

Existen dos grupos de información que debemos guardar: las distancias entre las islas y los nombres de las mismas.

Para almacenar las distancias interinsulares debemos notar que estas dependen de dos variables: de una isla de origen y otra de destino. Como ya se dijo, cuando varios datos dependen de dos variables, debemos pensar en almacenarlos en una matriz. En este caso usaremos una matriz de 5 x 5. Cada fila, al igual que cada columna, representará a una de las cinco islas. Entonces podríamos definir:

```
int Dist[5][5];
```

La idea detrás de esta definición es poder almacenar en el elemento `Dist[i][j]` la distancia entre la i -ésima isla y la j -ésima isla. Obviamente no es claro cuál sería la i -ésima o la j -ésima isla. Tendríamos que definir previamente un orden arbitrario. Para ello podríamos hacer:

```
enum islas {stcruz, floreana, baltra, isabela, cristobal};
```

Esto último facilitará la lectura del código. Ahora, viendo el mapa, ya se podría escribir:

```
Dist[stcruz][cristobal]=86;
```

para indicar que la distancia de la isla de Santa Cruz a San Cristóbal es 86 Km. Puedo repetir el proceso para cada dato del mapa mostrado. Dado que la distancia desde A hasta B es la misma que hay desde B hasta A, al final terminaremos declarando una matriz simétrica.

En cuanto a los nombres de las islas uno podría almacenarlas en:

```
char* nombres[5];
```

Luego, aprovechando la enumeración definida anteriormente, podríamos hacer asignaciones como:

```
nombres[stcruz] = "Isla Santa Cruz";
```

El programa completo quedaría de la siguiente forma:

```
#include <stdio.h>
enum islas {stcruz, floreana, baltra, isabela, cristobal};

int main(void) {
    char* nombres[5];
    nombres[stcruz] = "Isla Santa Cruz";
    nombres[floreana] = "Isla Floreana";
    nombres[baltra] = "Isla Baltra";
    nombres[isabela] = "Isla Isabela";
    nombres[cristobal] = "Isla San Cristobal";
    int Dist [5][5] = {0};
    Dist[stcruz][floreana] = Dist[floreana][stcruz] = 75;
    Dist[stcruz][isabela] = Dist[isabela][stcruz] = 91;
    Dist[stcruz][baltra] = Dist[baltra][stcruz] = 45;
    Dist[stcruz][cristobal] = Dist[cristobal][stcruz] = 86;
}
```

La declaración `int Dist [5] [5] = {0}` inicializa la matriz de distancias interinsulares con ceros. Luego de asignar los 4 datos conocidos, los ceros de las celdas restantes podrían interpretarse como rutas no disponibles.

Al final del `main` el lector puede añadir unas líneas para validar la información almacenada en `Dist`. Para ello se podría utilizar sentencias como:

```
printf("Distancia de %s a %s=%d", nombres[stcruz], nombres[baltra],
       Dist[stcruz][baltra]);
```

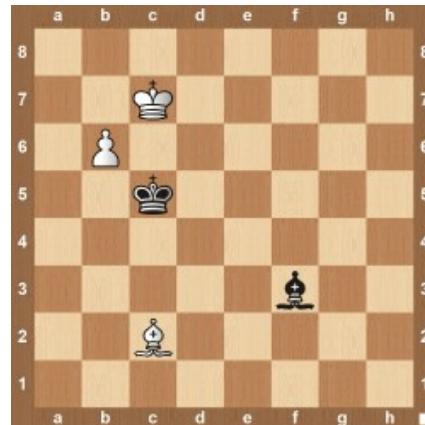
Note que definir una enumeración no nos obliga a tener que declarar variables de esa enumeración. Como en este ejercicio, uno podría declarar una enumeración solamente para sacar provecho de las constantes que automáticamente se ponen a disposición del programador.

6. Buscando a Bobby Fischer

Los tableros de ajedrez se dividen en filas (1-8) y columnas (a-h). Así se establece un sistema de coordenadas que nos permite hacer referencia a cada casilla.

En cuanto a las piezas, estas pueden ser de diferentes tipos: peones, torres, caballos, alfiles, damas y reyes. Tanto las piezas como las casillas del tablero pueden ser negras o blancas.

Se le pide representar el estado actual de la partida mostrada como una matriz.



Solución:

Es obvio que la matriz a definir debe tener 8 filas y 8 columnas. Pero no es tan obvio decidir el tipo de la matriz. Una matriz de caracteres o números flotantes, por ejemplo, no sería suficiente. ¿Cómo guardaríamos el dato de que en la casilla c5 hay un rey negro?, ¿qué carácter o número flotante usaríamos para representar un rey negro, un alfil blanco, o una casilla vacía?

Empezaremos definiendo un tipo de datos `Pieza`, que permita encapsular dos atributos: el tipo y el color de una pieza.

Tanto el tipo como el color de la pieza sólo pueden tomar unos pocos valores; por ello, consideramos conveniente definir primero las siguientes enumeraciones:

```
typedef enum tipo_pieza {
    vacio=0, peon, torre, caballo, alfil, dama, rey
} Tipo;

typedef enum color_pieza {
    blanco, negro
} Color;
```

A la enumeración `tipo_pieza` se la ha dado el alias `Tipo`; a `color_pieza`, `Color`. Dentro de los posibles valores de pieza se ha incluido el enumerador `vacio=0` para representar las casillas vacías. Ahora se define la estructura `Pieza`, cuyos atributos serán los tipos enumerados ya declarados:

```
typedef struct pieza {
    Tipo tp;
    Color cl;
} Pieza;
```

Nuestro tablero ya podría ser definido como `Pieza tablero[8][8]`. Siendo así, la asignación del rey blanco en la casilla c7, por ejemplo, puede implementarse como:

```
tablero[1][2]=(Pieza){rey, blanco};
```

Las ideas expuestas hasta aquí son suficientes para que podamos representar el juego mostrado como una matriz de Pieza's. El único inconveniente sería que tendríamos que asignar explícitamente cada casilla del tablero, incluso a las casillas vacías. Una manera de evitar este tedioso trabajo es inicializando el tablero con:

```
Pieza tablero[8][8]={0};
```

Lo anterior hace que todos los bits reservados para la matriz `tablero` sean inicializados en cero, lo que implica que el atributo `tablero[i][j].tp` será inicializado en cero (vacío) para todo `i` y `j`. Teniendo un tablero lleno de casillas vacías, sólo nos faltaría colocar las cinco piezas mostradas en sus casillas correspondientes.

El programa final quedaría como sigue:

```
#include <stdio.h>

typedef enum tipo_pieza {
    vacio=0, peon, torre, caballo, alfil, dama, rey
} Tipo;

typedef enum color_pieza {
    blanco, negro
} Color;

enum columnas {ca=0, cb, cc, cd, ce, cf, cg, ch};
enum filas {f8=0, f7, f6, f5, f4, f3, f2, f1};

typedef struct pieza {
    Tipo tp;
    Color cl;
} Pieza;

int main(void) {
    Pieza tablero[8][8] = {0};
    tablero[f2][cc] = (Pieza) {alfil, blanco};
    tablero[f5][cc] = (Pieza) {rey, negro};
    tablero[f6][cb] = (Pieza) {peon, blanco};
    tablero[f7][cc] = (Pieza) {rey, blanco};
    tablero[f3][cf] = (Pieza) {alfil, negro};
}
```

Las enumeraciones `enum columnas` y `enum filas` fueron definidas para no tener que convertir mentalmente las coordenadas del tablero en coordenadas de matriz. En lugar de calcular que la casilla b6 es en realidad `tablero[2][1]`, pensamos que resulta más mnemotécnico usar `tablero[f6][cb]`, donde f6 se refiere a la fila 6 y cb a la columna b.

7. Horario de clases

Se le pide representar en un arreglo de variables la información mostrada en el siguiente horario de clases. Las celdas coloreadas permiten conocer el curso dictado, el aula y la duración de varias reuniones.

Hora inicio	LUN	MAR	MIE	JUE	VIE
8:00	HISTORIA		MUSICA C3		HISTORIA B2
9:00	B1				
10:00			MUSICA C2		PINTURA A4
11:00					
12:00					
13:00		HISTORIA C1			

Solución:

Podemos representar el horario como un arreglo de reuniones. Cada reunión puede ser implementada con una estructura del siguiente tipo:

```
enum dia {Lun, Mar, Mie, Jue, Vie};

typedef struct clase {
    char *curso;
    enum dia dia_clase;
    char hora_inicio;
    char duracion;
    char *aula;
} Reunion;
```

El tipo `Reunion` permitirá almacenar el curso dictado, el día, hora, duración y aula de cada reunión. En el atributo `curso` se guardará una cadena de texto; en el atributo `dia_clase`, alguna de las constantes: `Lun`, `Mar`, ..., `Vie`, definidas en la enumeración `enum dia`; en `hora_inicio` y `duracion` se guardan dos enteros en el rango 8-13 y 1-6, respectivamente. Recuerde que el tipo `char` también permite guardar enteros pequeños. Finalmente, `aula` almacena otra cadena texto, el nombre del aula.

Para almacenar todas las reuniones podemos definir el arreglo:

```
Reunion rs[6];
```

y luego podemos asignar cada reunión de dos posibles formas. En una sola línea:

```
rs[0] = (Reunion) {"Historia", Lun, 8, 2, "B1"};
```

o asignando individualmente cada atributo:

```
rs[5].curso ="Pintura";
rs[5].dia_clase = Vie;
rs[5].hora_inicio = 10;
rs[5].duracion = 3;
rs[5].aula = "A4";
```

También sería bueno grabar los nombres de los días, haciendo:

```
char *ndias[5] = {"Lunes", "Martes", "Miércoles", "Jueves", "Viernes"};
```

pues esto nos permitiría imprimir mensajes como: El curso de Historia se dicta los Lunes de 8 a 10 a.m. en el aula B1, para cada una de las reuniones.

Para imprimir Lunes como parte del último mensaje puede usarse el especificador de formato %s sobre la expresión:

```
ndias[rs[0].dia_clase]
```

En expresiones como la anterior, donde se utiliza subíndices anidados (dos pares de corchetes, uno dentro del otro), el sistema evalúa primero el subíndice interior. En la sentencia anterior primero se evalúa la expresión rs[0].dia_clase, cuyo valor es 0, i.e. el valor del enumerador Lun; y luego se utiliza este valor para evaluar ndias[0], que mostrará el primer elemento del arreglo ndias, o sea Lunes.

Para imprimir la hora de finalización de una reunión se puede usar el especificador %d directamente sobre la expresión rs[0].hora_inicio + rs[0].duracion. No hay necesidad de declarar variables para almacenar la hora de finalización de cada reunión.

Se deja al lector que integre en un programa todos los fragmentos de código mostrados.

PROBLEMAS PROPUESTOS

1. Escriba un programa que solicite al usuario un número en base hexadecimal y que le muestre su representación en base decimal.
2. Escriba un programa que solicite el ingreso de un número decimal (p.ej. 3.14) y que muestre por separado su parte entera y su parte decimal (p.ej. "La parte entera es 3 y la parte decimal es 0.14").
3. Escriba un programa que solicite el ingreso del número de patas y cabezas que hay en una granja de patos y conejos. El programa debe mostrar el número de animales de cada tipo.
4. Crear un programa que pida al usuario dos datos, hh y mm, que representen la hora actual (p.ej. hh=20 y mm=15 representa las 08:15 pm). El programa debe añadir un minuto a la hora ingresada y mostrar el resultado en pantalla (p.ej. "En un minuto serán las 20:16").
5. Crear un programa que pida al usuario los coeficientes a, b y c de la ecuación $ax^2+bx+c=0$ y muestre las soluciones de la misma. Asuma que las soluciones siempre son reales.
6. Crear un programa que pida al usuario dos ecuaciones, $a_1x+b_1y=c_1$ y $a_2x+b_2y=c_2$, y muestre las soluciones x y y.
7. En el problema *Tras los pasos de Darwin* las distancias interinsulares se almacenaron en una matriz de números. Se le pide generalizar este programa para que también se almacenen los tiempos de viaje (min) de una isla a otra. Considere que los medios de transporte disponibles (p.ej. yate, lancha, avión) no son los mismos entre cada par de islas y, por lo tanto, no debe asumirse que el tiempo de viaje es proporcional a las distancias.
8. En el problema *Buscando a Bobby Fischer* se representó un tablero de ajedrez como una matriz de piezas. Modifique el programa de modo que ahora pueda ser representado como un arreglo.
9. En el problema del horario de clases, este fue representado como un arreglo de reuniones. Modifique el programa para que el horario pueda ser representado como un arreglo de cursos. Asuma que cada curso tiene como máximo tres reuniones semanales.

CAPÍTULO

2

PROGRAMACIÓN ESTRUCTURADA

La programación estructurada es una técnica que permite implementar programas fáciles de leer, comprender y modificar. Exige que el programador escriba su código utilizando solamente tres constructos fundamentales: secuencia, condición e iteración.

En C, la implementación de instrucciones secuenciales es trivial. Para especificar que una instrucción, `i1`, debe ser ejecutada antes que otra, `i2`, sólo se requiere escribir estas instrucciones una a continuación de la otra, separadas por un punto y coma, p.ej. `i1; i2`.

C también permite implementar instrucciones condicionales; es decir, nos permite informar al sistema que ciertas instrucciones sólo deben ser ejecutadas bajo ciertas condiciones, pero no necesariamente siempre. Para esto el lenguaje ofrece tres mecanismos: la sentencia `if-else`, la sentencia `switch` y el operador ternario.

Finalmente, para no tener que reescribir un mismo conjunto de instrucciones varias veces, el lenguaje C nos permite especificar que dichas instrucciones deberían ser ejecutadas iterativamente, una y otra vez, hasta que se cumpla una determinada condición. El lenguaje ofrece tres maneras alternativas de implementar esta prescripción: el bucle `for`, el bucle `while` y el bucle `do-while`.

El resto del capítulo se dividirá en dos secciones en las que estudiaremos separadamente las sentencias condicionales y las sentencias iterativas.

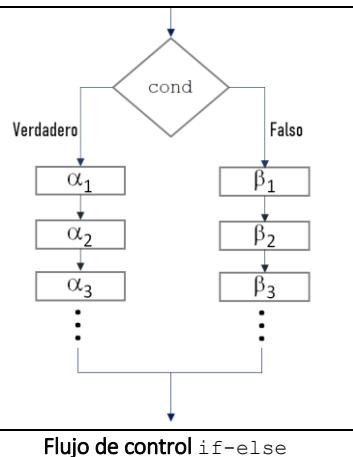
SENTENCIAS CONDICIONALES

Un programa es una secuencia de instrucciones. Pero esto no significa que cada instrucción de un programa debe ser ejecutada siempre, incondicionalmente. Muchas veces es conveniente asociar condiciones a ciertas instrucciones, de modo que estas sólo se ejecuten cuando se cumplan dichas condiciones.

Esta sección estudiará las tres formas de implementar condiciones en C.

SENTENCIA IF-ELSE

La forma más utilizada de implementar la condicionalidad en el lenguaje C es a través de las sentencias `if-else`, cuya sintaxis se muestra a continuación:

<pre>if (cond) { // bloque if α₁; α₂; α₃; ... } else { // bloque else β₁; β₂; β₃; ... }</pre>	
---	---

El bloque de código anterior se lee así: «Si se cumple la condición `cond`, se ejecutarán las instrucciones $\alpha_1; \alpha_2; \alpha_3; \dots$; de lo contrario, se ejecutarán las instrucciones $\beta_1; \beta_2; \beta_3; \dots$ ».

El grupo de instrucciones $\alpha_1; \alpha_2; \alpha_3; \dots$ junto con las llaves que las delimitan se conoce como **bloque if**. Análogamente, el bloque $\{\beta_1; \beta_2; \beta_3; \dots\}$ se denomina **bloque else**. La línea `if (cond)` suele llamarse **punto de bifurcación**, porque en este punto el flujo de control se desvía, hacia el bloque if o hacia el bloque else. Sólo uno de estos dos bloques será ejecutado dependiendo del valor de verdad de la condición `cond`.

La condición `cond` suele ser una expresión lógica (también llamada expresión booleana), aunque también podría ser una variable, p.ej. `cont`, o una constante, p.ej. `100`, `true`, etc.

Una **expresión lógica** es una serie de comparaciones ($<$, $>$, $<=$, $>=$, $==$, $!=$) conectadas por medio de operadores lógicos, AND (`&&`), OR (`||`) y NOT (`!`), p.ej. `edad > 10 && edad <= 20`. El resultado de estas expresiones siempre es un valor de verdad: verdadero o falso.

En el diagrama de flujo mostrado (lado derecho del gráfico anterior), las flechas indican la dirección del flujo de control; es decir, el orden en que se ejecutarán las instrucciones del programa. Los rectángulos representan las instrucciones del programa; los rombos indican una toma decisiones. En cada toma de decisiones el flujo de control se desvía hacia un lado u otro, hacia las instrucciones del bloque if o del bloque else.

El siguiente programa, `peso1.c`, servirá de ejemplo para ilustrar lo dicho:

```
#include <stdio.h>
int main(void) {
    int peso;
    float talla;
    puts("Ingrese su peso:"); scanf("%d", &peso);
    puts("Ingrese su talla:"); scanf("%f", &talla);

    float IMC = peso/(talla*talla);
    if (IMC>18.5 && IMC<24.9) {
        printf("Peso saludable");
    }
    else {
        printf("Tiene problemas con su peso");
    }
}
```

peso1.c

Este programa informa al usuario sobre el estado de su peso. Para ello calcula su índice de masa corporal (IMC) a partir de su peso y su talla. Ambos datos son leídos desde el teclado para luego ser almacenados en sendas variables.

Dependiendo del IMC calculado el programa nos dirá si nuestro peso es saludable o si tenemos un problema de peso. Sería absurdo que el programa nos dijera ambas cosas a la vez; tiene que ser una u otra, o estamos saludables o no lo estamos. Por ello, las sentencias contradictorias `printf("Peso saludable")` y `printf("Tiene problemas con su peso")` están separadas, una dentro del bloque if y la otra en el bloque else, para que así, en cada ejecución del programa, sólo se ejecute una u otra, pero nunca las dos.

La sentencia `printf("Peso saludable")` se ejecuta sólo si la condición `IMC>18.5 && IMC<24.9` es verdadera; o sea, cuando el índice de masa corporal está en el rango (18.5, 24.9). De lo contrario se ejecutan las instrucciones del bloque else, i.e. `printf("Tiene problemas con su peso")`.

En el ejemplo anterior tanto el bloque if como el bloque else sólo contienen una instrucción; pero, en general, podrían contener muchas más. No existe ningún límite sobre la cantidad de instrucciones que podemos colocar dentro del bloque if o el bloque else. Lo que sí existe es una regla que permite omitir las llaves, {}, de un bloque (if o else) que contiene una única instrucción. Esto significa que el programa anterior seguirá comportándose igual si eliminamos las llaves del bloque if y del bloque else.

Es importante saber que la presencia del bloque else siempre es opcional. Muchas veces sólo necesitamos especificar instrucciones como: «Si esta condición se cumple entonces debe ejecutarse este bloque de instrucciones», sobreentendiéndose que si la condición no se cumple simplemente no se

debería hacer nada. Para este tipo de prescripciones basta con usar un bloque if. Por ejemplo, si eliminásemos el bloque else del código anterior, el nuevo programa, sin importar cuántos valores distintos asignemos a peso y talla, sólo imprimiría "Peso saludable" cada vez que el IMC fluctúe entre 18.5 y 24.9; y no haría nada en otros casos.

Finalmente, es hora de admitir que la única razón por la que usamos printf en el programa anterior se debe a que es la única función de impresión que hemos estudiado hasta el momento. En adelante, el uso de printf se limitará a aquellas situaciones que requieran imprimir valores de variables, o sea, cuando la cadena de formato de printf incluya algún identificador. Para imprimir cadenas constantes como "Peso saludable" (que no contienen %d, %f, etc.), se usará la función puts. Con toda confianza, cambie printf por puts en el programa anterior y vuelva a ejecutarlo.

SENTENCIAS if-else ANIDADAS

Es posible introducir una sentencia if-else dentro de otra, y así hasta al infinito. El anidamiento de sentencias if-else permite desviar el flujo de control muchas veces hasta alcanzar un bloque de código específico, cuya ejecución implicaría el cumplimiento de una serie de condiciones previas ya satisfechas.

Para explicar lo dicho refinemos el burdo programa anterior, pesol.c. Este sólo identificaba problemas de peso, pero no daba más detalles al respecto; no se podía saber si se trataba de un problema de falta de peso o de sobrepeso. En cambio, el programa peso2.c que se muestra abajo es más específico.

Por ahora concentrémonos en la parte final del programa. Ahora existe una sentencia if-else dentro de nuestro antiguo bloque else. Antes, el bloque else era alcanzado cada vez que se violaba la condición IMC>18.5 && IMC<24.9, en cuyo caso el programa hacía una sola cosa y siempre la misma cosa: imprimir el mensaje: Tiene problemas con su peso. Ahora, dentro del bloque else se realiza una segunda evaluación del IMC, y con ello podemos acondicionar nuestro programa para que pueda realizar otras dos nuevas tareas, dependiendo del resultado de la segunda evaluación.

```
#include <stdio.h>
int main(void) {
    int peso;
    float talla;

    puts("Ingrese su peso:"); scanf("%d", &peso);
    puts("Ingrese su talla:"); scanf("%f", &talla);

    float IMC = peso / (talla*talla);

    if (IMC > 18.5 && IMC < 24.9)
        puts("Peso saludable");
    else {
        if (IMC <= 18.5)
            puts("Tiene falta de peso");
        else
            puts("Tiene sobrepeso");
    }
}
```

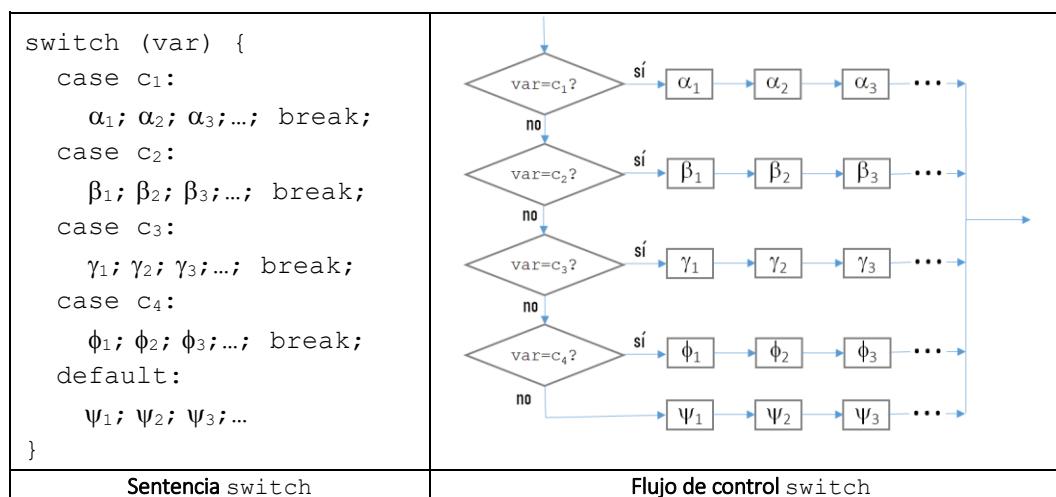
peso2.c

Si luego de que la condición `IMC > 18.5 && IMC < 24.9` es violada, se cumple que `IMC<=18.5`, entonces el problema sería uno de falta de peso, y este será el mensaje que se imprimirá en pantalla. En cambio, si luego de que la condición `IMC>18.5 && IMC<24.9` es violada tampoco se cumpliere que `IMC<=18.5` (lo cual implicaría que `IMC>=24.9`), entonces estaríamos ante el hoy frecuente problema de sobrepeso y así se le haría saber al usuario.

SENTENCIA SWITCH

Cada vez que se ejecuta una sentencia `if-else` el sistema tiene que evaluar una condición booleana arbitrariamente compleja. En cambio, la ejecución de una sentencia `switch` sólo involucra comparaciones de igualdad. Las sentencias `switch` son útiles cuando se necesita comparar una variable contra una serie de constantes y, definir, para cada posible caso de igualdad, el conjunto de instrucciones a ejecutar.

En la figura de abajo, el diagrama del lado derecho muestra que el código de la izquierda equivale a una serie de comparaciones de la variable `var` contra las constantes `c1, c2,...`. Estas comparaciones están representadas en los cuatro rombos del diagrama de flujo.



Abajo se muestra un programa donde se utiliza la sentencia `switch`. Este programa solicita al usuario la sección del curso de programación que está cursando para luego mostrar el nombre del profesor encargado de dicha sección. Para quienes lo conozcan con otro nombre, la sección es un identificador numérico que permite distinguir los múltiples grupos de estudiantes que podrían estar cursando una misma materia.

Al llegar a la sentencia `switch`, el sistema compara el valor de `seccion` con las constantes 101, 102, 103, 105 y 107, en ese orden. Si la variable `seccion` tuviese alguno de estos valores, se ejecutarían las sentencias que están al lado de la etiqueta que hace referencia a dicho valor. Por ejemplo, si el valor de `seccion` fuese 103, el programa ignorará las sentencias que acompañan a las etiquetas `case 101` y `case 102`, y sólo ejecutará las sentencias que acompañan a la etiqueta `case 103`; es decir, las

instrucciones `puts ("Cuenca")` y `break`. La primera imprimirá Cuenca en pantalla; la segunda hará que el programa ya no tenga que molestarse en seguir comparando `seccion` con 105, 107 y, en general, con toda la larga lista de constantes que podrían haber estado debajo de `case 103`. La sentencia `break` arroja el flujo de control del programa fuera del `switch`.

```
#include <stdio.h>
int main(void) {
    int seccion;
    puts("Ingrese su sección:");
    scanf("%d", &seccion);
    switch(sección)
    { case 101: puts("Ritchie Jr."); break;
      case 102: puts("Zuckerberg"); break;
      case 103: puts("Cuenca"); break;
      case 105: puts("Page"); break;
      case 107: puts("Brin"); break;
      default: puts("Sección no registrada");
    }
}
```

sección.c

También puede ocurrir que la variable `sección` tenga un valor distinto a todos los listados (101, 102, 103, 105 y 107). En ese caso se terminaría ejecutando las instrucciones que se encuentran a la derecha de la etiqueta `default`. En nuestro caso se imprimiría Sección no registrada. La etiqueta `default` es opcional; cuando esta se incluye, siempre al final del `switch`, no necesita una sentencia `break` que la acompañe, puesto que, de todos modos, nunca hay más etiquetas que evaluar después de esta.

Todo lo que puede implementarse con una sentencia `switch` también puede implementarse con sentencias `if-else` anidadas, tal como se muestra abajo. Lo opuesto no siempre es cierto.

<pre>switch(sección) { case 101: puts("Ritchie Jr."); break; case 102: puts("Zuckerberg"); break; case 103: puts("Cuenca"); break; case 105: puts("Page"); break; case 107: puts("Brin"); break; default: puts("Sección no registrada"); }</pre>	<pre>if(sección==101) puts("Ritchie Jr."); else if(sección==102) puts("Zuckerberg"); else if(sección==103) puts("Cuenca"); else if(sección==105) puts("Page"); else if(sección==107) puts("Brin"); else puts("Sección no existe");</pre>
--	--

Sentencia `switch` y `if-else` equivalentes

Las etiquetas de una sentencia `switch` no necesariamente deben tener instrucciones asociadas a estas. Un conocido truco que se desprende de lo anterior se observa en el siguiente fragmento de código:

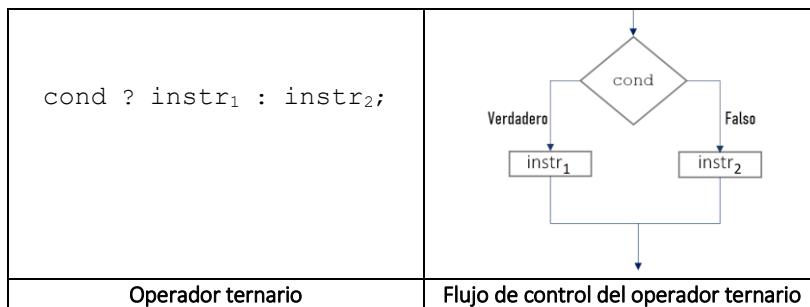
```
switch(seccion)
{ case 101:
    case 102: puts("Zuckerberg"); break;
    case 107:
    case 103: puts("Cuenca"); break;
    case 105: puts("Page"); break;
    default: puts("Sección no registrada");
}
```

Uso de etiquetas vacias

Con el código anterior el programa anunciará a Zuckerberg como docente a cargo de las secciones 101 y 102; a Cuenca, de las secciones 107 y 103; y a Page, de la sección 105.

OPERADOR TERNARIO

El operador ternario puede considerarse como una sentencia `if-else` minimalista, de una sola línea. La sintaxis es como sigue:



Se llama ternario porque involucra tres componentes: dos instrucciones, `instr1` y `instr2`, y una condición booleana, `cond`. La intimidante expresión `cond?instr1:instr2` al final siempre quedará reducida a `instr1`, cuando `cond` sea verdadera, o a `instr2`, cuando `cond` sea falsa. A continuación mostramos un programa donde se utiliza este operador:

```
#include <stdio.h>

int main(void) {
    int A, B;
    puts("Ingrese A:"); scanf("%d", &A);
    puts("Ingrese B:"); scanf("%d", &B);

    int mayor = A > B? A: B;
    int menor = A < B? A: B;

    printf("El número mayor es %d\n", mayor);
    printf("El número menor es %d\n", menor);
}
```

ternario.c

Para analizar el código anterior asumamos que los valores que el usuario ingresa por teclado para A y B son tales que B es mayor que A. En ese caso, la condición A > B sería falsa y, por lo tanto, la asignación mayor = A>B ? A:B acabaría convirtiéndose en mayor=B. Toda la expresión A>B ? A:B termina reduciéndose a B a causa de la falsedad de A>B.

Análogamente, siguiendo con la misma suposición, la condición A < B sería verdadera, lo que convertiría la engorrosa expresión menor = A<B ? A:B en la sencilla menor=A.

VARIABLES Y CONSTANTES COMO EXPRESIONES BOOLEANAS

Finalmente explicaré algo que ya mencioné cuando estudiamos el bloque `if-else` y que mis atentos lectores deben recordar muy bien. Entonces dije que las condiciones suelen ser expresiones booleanas pero que también podían ser variables o constantes.

Cuando se utiliza una variable o una constante en lugar de una condición booleana el sistema evalúa la representación interna de dicha variable/constante; es decir, los bits que se utilizan para codificar dicho dato. Si todos estos bits son ceros, la variable/constante será evaluada como falso; de lo contrario, será considerada como verdadero.

Para ilustrar lo anterior comentaré el siguiente programa:

```
#include <stdio.h>
int main(void) {
    int var;
    var = 0;
    var? puts("aquí"): puts("acá"); //imprime acá
    var = 9741;
    var? puts("aquí"): puts("acá"); //imprime aquí
}
```

ternario2.c

Este código imprimirá primero acá y luego aquí. Esto es así porque cuando var = 0, su representación interna es una retahíla de 32 bits (4 bytes) de puros ceros. Siendo así, el valor de var es considerado como falso y la expresión var? puts("aquí"): puts("acá") se reduciría a puts("acá"). Luego, unas líneas más abajo, cuando se hace var = 9741, la representación interna de var ahora sí contiene algunos bits encendidos; es decir, algunos 1's (convierta 9741 a binario y vea estos bits con sus propios ojos), y, por ende, ahora var sería considerada como verdadero. Consecuentemente, la última línea imprime aquí.

SENTENCIAS ITERATIVAS

C nos permite definir sentencias iterativas o bucles, grupos de instrucciones que deben ejecutarse repetidamente hasta que se cumpla o se deje de cumplir cierta condición. Hay tres maneras equivalentes de implementar bucles. La más usada es probablemente la sentencia `for`.

BUCLE FOR

La sintaxis para definir un bucle for es la siguiente:

<pre>for(inic; cond; act) { /* bloque for */ α_1; α_2; α_3; ... }</pre>	
Bucle for	Flujo de control del bucle for

La sentencia anterior indica que las instrucciones α_1 ; α_2 ; α_3 ;... deben ejecutarse repetidamente siempre que la condición booleana `cond` sea verdadera.

La primera sección de la cabecera `for`, `inic`, se llama **sección de inicialización**. Aquí se declaran variables que, por lo general, forman parte de la **condición de parada** `cond`. La tercera sección, `act`, es la **sección de actualización**. Aquí normalmente se actualizan algunas de las variables utilizadas en `cond`. Esto suele alterar el valor de verdad de `cond`, lo que, eventualmente, detendría el bucle.

Siendo más precisos, cuando el flujo de control llega a la sentencia `for` ocurre lo siguiente:

1. Se ejecuta la sección de inicialización `inic`.
2. Se verifica la condición `cond`. Si esta verdadera, se ejecutan las instrucciones α_1 ; α_2 ; α_3 ;.... Si es falsa, el bucle ha terminado; el programa continúa ejecutando las instrucciones que se encuentran por debajo, fuera del bloque `for`.
3. Se ejecuta la sección de actualización, `act`. Se vuelve al paso 2.

De lo anterior, el siguiente código imprime los números enteros de 1 a 10:

```
for(int number = 1; number <= 10; number++)
    printf("%d", number);
```

Imprimiendo 1, 2, 3,..., 10

Inicialmente se define la variable `number` con el valor 1. Luego se verifica la condición `number<=10`, que, en ese momento, es verdadera. Por ello, el programa ejecuta las instrucciones del bloque `for` –la sentencia `printf`, en nuestro caso–, lo que causa que se imprima el número 1 en pantalla. Luego se ejecuta la sentencia de actualización, `number++`. Esto incrementa el valor de `number` en una unidad. En ese momento el programa vuelve a verificar la condición `number<=10`. A pesar que `number` ya tiene el valor 2 la condición sigue siendo cierta. Por ello se realiza una nueva iteración, de la que resulta la impresión del 2 en pantalla y, posteriormente, el incremento de `number` a 3.

El programa sigue la misma lógica durante varias iteraciones. Sin tener que ser exhaustivos se puede prever que eventualmente la variable `number` alcanzará el valor de 10. Ahí todavía será válida la condición `number<=10` y, por lo tanto, se imprimirá 10 en pantalla y se incrementará `number` a 11. Por fin, en ese momento y por primera vez, la condición `number<=10` ya no será cierta. Ahí terminará el bucle. Los números de 1 a 10 ya han sido impresos.

Hay dos detalles adicionales que vale la pena mencionar. Primero, cuando el bloque `for` consta de una sola sentencia no es necesario delimitarlo con llaves. Por ello hemos omitido las llaves que podrían haber encerrado a la sentencia `printf`. Segundo, la variable `number`, declarada dentro de la sección de inicialización, sólo es reconocible dentro del bucle, i.e. en la cabecera y en el bloque `for`. Si usaramos `number` fuera del bucle nos encontraríamos con un error de compilación: «variable no declarada».

SUMADORES Y CONTADORES

Un uso muy común de los bucles consiste en recorrer arreglos. Por ejemplo, el programa mostrado abajo recorre un arreglo `notas` para calcular el promedio de los alumnos que han aprobado un determinado curso. En este caso el arreglo ya contiene unos valores predeterminados, pero esto no afectará las explicaciones que queremos ofrecer.

Como se sabe, para calcular el promedio de los aprobados se debe determinar: (a) el número de aprobados y (b) la suma total de todas sus notas, para luego dividir esta cantidad entre aquella. Las variables `num` y `total` han sido declaradas para que puedan almacenar los dos valores antes mencionados.

```
int main(void) {
    float notas[]={6.1, 3.2, 3.7, 8.9, 5.6, 8.1, 9.2, 10, 2.8};

    int size = sizeof(notas)/sizeof(notas[0]);
    float total=0.0;
    int num=0;

    for(int i=0; i<size; i++) {
        if(notas[i] > 5) {
            total += notas[i];
            num += 1;
        }
    }
    printf("El promedio es %f", total/num);
}
```

Calculando el promedio de los aprobados

Lamentablemente no es posible asignar los valores de `num` y `total` con una sola instrucción (a menos que uno quiera calcularlos manualmente y asignarlos directamente). El programa debe inspeccionar todas las notas, una por una, y, cada vez que se encuentre una nota aprobatoria (`notas[i]>5`), se debe incrementar el valor de la variable `num` (`num+=1`) y acumular dicha nota en la variable `total` (`total+=notas[i]`). Así, al final del bucle, la variable `num` terminaría conteniendo el número total de aprobados; y la variable `total`, la suma de todas las notas aprobatorias. Al igual que algunos colegas, suelo llamar **contador** a las variables que, como `num`, se incrementan de uno en uno dentro de un bucle con el objetivo de contar las ocurrencias de una situación específica. También suelo llamar **sumador** a las variables que, como `total`, van acumulando la suma de varios datos similares dentro de un bucle.

Para que la lógica descrita anteriormente pueda ser aplicada a cada elemento de `notas`, el valor de la variable `i` (usada en `notas[i]`) debe variar desde `0` hasta `size-1`, siendo `size` el tamaño del arreglo. Esta variación se especifica en la cabecera del bucle, `for (int i=0; i<size; i++)`.

BUCLES ANIDADOS

Es posible definir un bucle dentro de otro. Esta implementación aparece de manera natural cuando se trabaja con matrices. En estos casos, un bucle suele usarse para navegar por las filas de la matriz, y el otro, interior al primero (anidado), para visitar los elementos de la fila que está siendo visitada.

El siguiente código imprime los valores de una matriz `M` de 3×4 :

```
#include <stdio.h>
int main(void) {
    int M[3][4]={{1,2,3,4},
                 {5,6,7,8},
                 {9,0,1,2}};
    for(int i=0; i<3; i++) {
        for(int j=0; j<4; j++) {
            printf("%d\t", M[i][j]);
        }
        printf("\n");
    }
}
```

Imprimiendo una matriz

El bucle externo, `for (int i=0; i<3; i++)`, se ejecutará tres veces, para $i=0,1,2$. Para cada i se imprimen los elementos de la i -ésima fila y luego un salto de línea, con `printf("\n")`. Dado que cada fila posee 4 valores, la impresión de una fila no es trivial y requiere un segundo bucle, anidado dentro del primero, cuya cabecera es `for (int j=0; j<4; j++)`. En cada iteración del segundo bucle se imprime el j -ésimo elemento de la i -ésima fila, o sea `M[i][j]`, seguido de un carácter de tabulación para que exista un espacio entre un dato y el siguiente.

Note que las llaves usadas para delimitar el bucle interno podrían omitirse pues este contiene una única sentencia `printf("%d\t", M[i][j])`. En cambio, las llaves del bucle externo son obligatorias, pues este contiene dos sentencias: la sentencia `for(int j=0; j<4; j++)` y `printf("\n")`.

SALIR DE UN BUCLE (`break`)

Normalmente un bucle se ejecuta varias veces hasta que una condición de parada, especificada en la cabecera del bucle, deja de ser verdadera. Pero también existe una manera de salir abruptamente de un bucle, desde el cuerpo del mismo. Para que ello ocurra debe utilizarse la palabra reservada `break`, tal como se aprecia en el código mostrado abajo.

El programa solicita al usuario el ingreso de un número `N` y determina si dicho número es primo o no. La idea central detrás del programa consiste en dividir `N` entre `2, 3, 4,..., N-1`. Si alguna de estas divisiones fuese exacta, esto significaría que `N` tiene algún divisor distinto de la unidad y de si mismo y, por lo tanto, ya no sería un número primo.

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

int main(void) {
    int N;
    bool primo = true;

    puts("Ingrese un número:");
    scanf("%d", &N);

    // verificar si N es primo
    for(int num=2; num<N; num++) {
        if(N % num == 0) {
            primo = false;
            break;
        }
    }

    primo ? printf("%d es primo", N) :
           printf("%d no es primo", N);
}
```

Validando si un entero es primo

La variable `num` se usa para dividir a `N`. Dentro del bucle `for(int num=2; num<N; num++)`, la variable `num` va tomando los valores `2, 3, 4,..., N-1`. En cada iteración se comprueba si `num` es un divisor de `N`, lo cual es cierto cuando `N % num == 0`. Apenas encontramos un `num` que sea divisor de `N` ya no tendría sentido seguir buscando más divisores. En ese mismo momento ya podríamos afirmar que `N` no es primo. Lo óptimo sería salir del bucle inmediatamente con la sentencia `break`. Y así lo hace el programa.

Luego de salir del bucle el programa ejecuta el operador ternario. Es aquí donde se imprime el mensaje que indica si el número `N` es primo o no. La impresión de un mensaje u otro depende del valor de una variable booleana llamada `primo`. Por conveniencia el programa empieza asumiendo que el usuario

ingresará un número primo (`primo=true`), pero apenas se detecta un divisor de N la asunción queda descartada (`primo=false`).

El tipo de datos `bool` no es un tipo nativo. Para utilizarlo se debe incluir el archivo `stdbool.h`.

CONTINUAR CON LA SIGUIENTE ITERACIÓN (`continue`)

Cuando trabajamos con bucles existe la posibilidad de abandonar la ejecución de un bloque de instrucciones a la mitad, por decirlo de algún modo, para comenzar con la ejecución de una nueva iteración sin haber concluido la anterior. El siguiente ejemplo muestra lo que acabamos de comentar:

```
for(int number=1; number <= 10; number++) {
    if(number % 2 == 0)
        continue;

    printf("%d ", number);
}
```

Imprimiendo 1,3,5,...,9

A falta de una imaginación más fecunda se ha hecho un pequeño cambio al primer programa que estudiamos en esta sección, el que imprime los números de 1 a 10. Ahora, dentro del bucle existe una sentencia `if`. Cada vez que `number` es par se ejecuta la instrucción `continue`. Esto hace que el flujo de control se dirija al final del bloque `for`, saltándose la sentencia `printf`, para iniciar una nueva iteración con el valor de `number` incrementado en una unidad.

Note la diferencia con la instrucción `break`. Esta permitía salir del bucle; en cambio, con `continue` sólo se abandona la iteración actual, no todo el bucle.

Dado que la sentencia `printf` será saltada para todos los números pares, el programa sólo imprimirá los números impares del intervalo 1,...,10; es decir, 1, 3, 5, 7, 9. En este ejemplo, la función `continue` se «saltó» una sola sentencia (`printf`); pero, en general, pudieron ser muchas más.

Para concluir, debe decirse que ninguna de las tres secciones que componen la cabecera del bucle `for` es obligatoria. Los tres fragmentos de código que se muestran a continuación prueban lo dicho. Cada uno de estos imprime los números de 1 a 10 de tres maneras distintas:

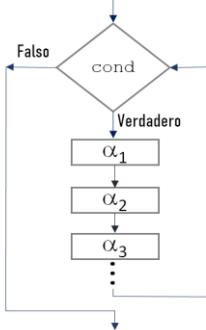
```
int i = 1;
for ( ; i <= 10 ; i++)
    printf("%d", i);
```

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

```
int i = 1
for ( ; ; ) {
    printf("%d", i++);
    if(i>10) break;
}
```

BUCLE WHILE

La sintaxis para definir un bucle while es la siguiente:

<pre>while(cond) { /* bloque while */ α₁; α₂; α₃; ... }</pre>	
Bucle while	Flujo de control del bucle while

Cuando el flujo de control alcanza el bucle while se llevan a cabo los siguientes pasos:

1. Se evalúa la condición cond. Si es verdadera, se ejecutan las instrucciones $\alpha_1; \alpha_2; \alpha_3; \dots$. En caso contrario, se termina el bucle y el programa continúa con la siguiente instrucción que se encuentra fuera del bloque while.
2. Se retorna al paso 1.

El siguiente programa calcula el factorial de un número N ingresado por teclado:

```
#include <stdio.h>

int main(void) {
    int fact = 1;
    int num = 2;
    int N;

    puts("Ingrese N: ");
    scanf("%d", &N);

    while(num <= N) {
        fact = fact * num;
        num++;
    }

    printf("%d ", fact);
}
```

Calculando el factorial de N

La idea central de este programa consiste en guardar dentro de la variable fact los productos acumulados $1 \times 2, 1 \times 2 \times 3, \dots$, hasta $1 \times 2 \times 3 \times \dots \times N$. Cada una de estas multiplicatorias será asignada a fact en sucesivas iteraciones del bucle while.

Dado que fact y num son inicializadas con 1 y 2, respectivamente, en la primera iteración fact tomará el valor 1×2 . Luego de ello, el valor inicial de num será incrementado en una unidad (num++), o sea a 3.

En la segunda iteración la variable fact será actualizada. Su nuevo valor será $1 \times 2 \times 3$ pues ese es el resultado de multiplicar su valor vigente, 1×2 , por num, que en ese momento vale 3. Así es como debe

leerse la sentencia `fact=fact*num`: «el nuevo valor de `fact` es igual a su valor vigente multiplicado por `num`».

De la condición de parada se prevé que `num` eventualmente tomará el valor de `N`, el dato ingresado por el usuario. En dicho momento `fact` ya tendría el valor $1 \times 2 \times 3 \times \dots \times (N-1)$ y, al ser ahora multiplicado por `num`, `fact` terminaría siendo actualizada al valor buscado, $1 \times 2 \times 3 \times \dots \times N$.

Después de esto la variable `fact` ya no será actualizada. El valor de `num` se incrementará en una unidad y sobrepasará el valor de `N`. La condición `num<=N` se hará falsa y el bloque `while` ya no volverá a ejecutarse. Fuera del bloque `while`, la variable `fact` será impresa con el último valor que le fue asignado.

Otro ejemplo ilustrativo del bucle `while` es la impresión de cadenas, que se muestra abajo:

```
char cad[50] = "Hola";
int i=0;

while(cad[i])
    putchar(cad[i++]);
```

El fragmento de código anterior imprime el contenido de `cad`, el mensaje `Hola`.

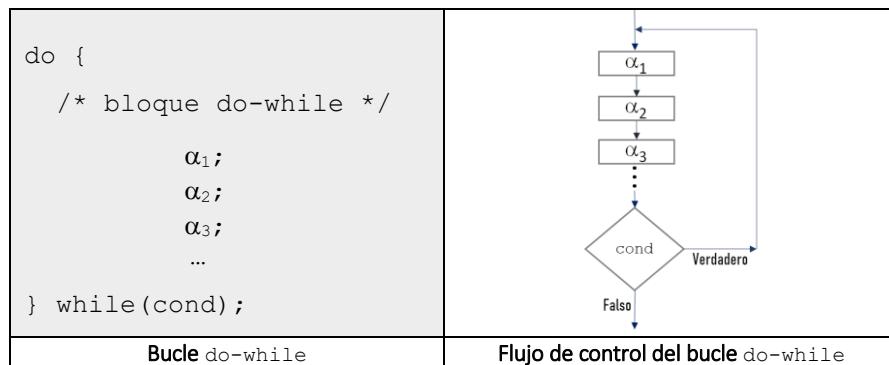
En este ejemplo el bloque `while` contiene una sola instrucción que, por ello mismo, no necesita ir entre paréntesis.

La función `putchar` (con prototipo en `stdio.h`) imprime el carácter pasado como argumento. En este caso, la sentencia `putchar(cad[i++])` hace que se imprima el `i`-ésimo carácter de `cad`, luego de lo cual se incrementa el valor de `i`. Dado que `i` inicia en cero, primero se imprimirá `cad[0]`, seguidamente `cad[1]`, y así sucesivamente.

El bucle terminará cuando la expresión `cad[i]` sea falsa. Esto ocurre cuando `i=4` puesto que `cad[4]` contiene al carácter de terminación '`\0`' que, internamente, se representa con 8 bits nulos.

BUCLE DO-WHILE

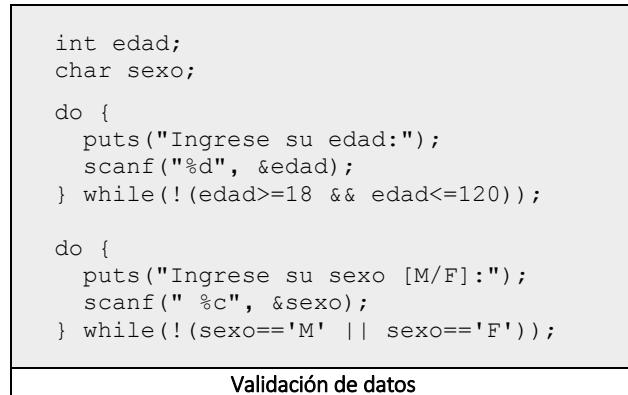
La sintaxis para definir un bucle do-while es la siguiente:



Cuando el flujo de control alcanza el bucle do-while se llevan a cabo los siguientes pasos:

1. Se ejecutan las instrucciones $\alpha_1; \alpha_2; \alpha_3; \dots$
2. Se evalúa la condición `cond`. Si es verdadera, se retorna al paso 1; en caso contrario, se termina el bucle y el programa continúa con la siguiente instrucción que se encuentra fuera del bloque do-while.

Esta estructura es muy útil para implementar sencillas validaciones de rango sobre los datos ingresados por el usuario, tal como se observa en el siguiente fragmento de código:



En el código anterior el programa no se limita a solicitar un dato y aceptar lo que al usuario se le ocurra ingresar, sino que verifica que el dato ingresado tenga un valor apropiado. De no ser así, el programa insistirá y volverá a solicitar que se reingrese el dato pedido.

El programa mostrado no concluirá hasta que el usuario haya ingresado (1) una edad que oscile entre 18 y 120 años, y (2) el carácter 'M' o 'F' como representación de su sexo (Masculino o Femenino).

El símbolo de negación, `!`, utilizado en la condición de parada del primer bucle sirve para indicar que la solicitud de edad deberá repetirse mientras no se ingrese una edad entre 18 o 120. El símbolo `!` niega

toda la expresión `edad>=18 && edad<=120`. Análogamente, la solicitud de ingreso de sexo debe ser formulada mientras el usuario no ingrese 'M' o 'F', los únicos valores aceptables. Nuevamente, el símbolo ! niega toda la expresión `sexo== 'M' || sexo== 'F'`.

Note el espacio en blanco en el especificador " %c". Este permite consumir los retornos de carro (ENTER) que podrían anteceder al carácter ingresado.

A continuación se muestra otro ejemplo del uso de `do-while`. En este nuevo ejemplo se simulan sucesivos lanzamientos de un par de dados. El programa culmina apenas se realiza un lanzamiento cuya suma de dados sea igual a 7.

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

int main(void) {
    int dado1;
    int dado2;

    do{
        dado1 = rand() % 6 + 1;
        dado2 = rand() % 6 + 1;
        printf("%d %d\n", dado1, dado2);

    } while(dado1 + dado2 != 7)
}
```

Simulando lanzamiento de dos dados

La simulación de un lanzamiento se implementa generando dos números enteros aleatorios entre 1 y 6. Cada uno de estos números estaría representando el resultado obtenido en un dado imaginario.

El programa imprimirá los resultados de los lanzamientos como una secuencia de pares de números, de modo que el último par siempre sumará 7.

Para generar un número aleatorio entre 1 y 6 hemos aprovechado la función `rand` (con prototipo en `stdlib.h`), que retorna un entero aleatorio. Dado que los enteros generados por `rand` no están calibrados en el rango 1-6, sino que son, por lo general, números mucho más grandes, debemos hacer un trabajo previo. Sin importar cuál sea el número devuelto por `rand()`, el residuo obtenido al dividir dicho número entre 6, `rand()%6`, tiene que estar necesariamente entre 0 y 5 (el residuo de una división entera siempre es menor que el divisor). Siendo así, la expresión `rand()%6+1` siempre devuelve un número aleatorio entre 1 y 6. Dicho número varía de iteración en iteración puesto que `rand()` está diseñado para retornar distintos números cada vez que se le invoca.

El programa simulará tantos lanzamientos como veces se ejecute el bucle `do-while`. Dado que el juego que estamos simulando termina cuando la suma de los dados lanzados es 7, la simulación sólo deberá ejecutarse mientras `dado1+dado2!=7`.

Antes de terminar nuestra discusión del bucle `do-while` note que los dos ejemplos anteriores, la validación de datos y la simulación de lanzamientos de dados, tienen algo en común, que podría darnos una pista sobre cuándo conviene utilizar este tipo de bucle. En ambos casos las instrucciones que van a repetirse deben ser ejecutadas por lo menos una vez. En el primer caso el programa tenía que pedir

datos al menos una vez; en el segundo, se tenía que hacer por lo menos un primer lanzamiento. A diferencia de los otros dos tipos de bucles, `for` y `while`, en el bucle `do-while` primero se hace «algo» y después se decide si conviene volver a hacer nuevamente ese «algo» o ya no.

Toda operación que puede implementarse con alguno de los tres tipos de bucle estudiados también puede implementarse con los otros dos restantes. Los tres tipos de bucles son computacionalmente equivalentes.

Finalmente, aunque las instrucciones `break` y `continue` sólo fueron mostradas para el bucle `for`, también pueden usarse dentro de los bucles `while` y `do-while`.

EL VIEJO GOTO

En los albores de la programación se solía utilizar extensamente la instrucción `goto`. Esta hace que el flujo de control del programa salte desde una línea del código hacia otra cualquiera. Entre muchas otras funcionalidades, la instrucción `goto` nos permite emular bucles sin necesidad de sentencias `for` o `while`. Sin embargo, a pesar de su versatilidad, muchos observaron que el uso excesivo de la sentencia `goto` terminaba produciendo programas cuyo código era enrevesado y difícil de comprender, código spaghetti.

En 1966, en su artículo «*Flow diagrams, turing machines and languages with only two formation rules*», Corrado Böhm y Giuseppe Jacopini lograron demostrar que la instrucción `goto` era totalmente prescindible, que cualquier programa podía ser implementado utilizando solamente tres constructos básicos: secuencia, condición e iteración. Ahí es donde nace la programación estructurada, aunque tuvo que pasar un buen tiempo para que los programadores se decidieran a adoptarla. El apoyo de algunos nombres importantes fue decisivo. En 1968, el prestigioso Edsger Dijkstra escribiría un manifiesto, «*Go To Statement Considered Harmful*», en el que disuadía a los programadores de usar la sentencia `goto` y los animaba a pasarse al bando de la programación estructurada.

A pesar del cargamontón desatado contra el `goto` hubo reacciones que defendieron su uso en situaciones específicas (p.ej. para salir de cadenas de bucles anidados). Lo cierto es que a la fecha el comando `goto` sigue siendo parte del lenguaje C aunque su uso nunca es recomendado.

Sin ánimos de revivir lo que ya ha sido enterrado, a continuación se presenta un programa que utiliza la sentencia `goto`. No se piense que lo hacemos embriagados de un romanticismo senil. Simplemente queremos que el joven lector tenga una buena idea de lo que la programación estructurada logró proscribir del lenguaje.

El programa mostrado abajo imprime la secuencia de números de 1 a 10 a pesar de no implementar ningún tipo de bucle.

Tal como se observa en el código, el uso de `goto` requiere definir etiquetas, como `loop` o `fin`. Estas sirven para indicar el punto exacto hacia donde debe redireccionarse el flujo de control.

```
#include <stdio.h>
int main(void) {
    int num = 1;
loop:
    printf("%d ", num);
    num += 1;
    if(num > 10)
        goto fin;
    goto loop;
fin:
    puts("Adios");
}
```

A pesar de no contener sentencias `for` o `while`, la instrucción `printf("%d ", num)` y la asignación `num+=1` se han ejecutado diez veces. Esto es posible gracias a `goto loop`, que retorna el flujo de control del programa hacia la etiqueta `loop`; es decir, hacia atrás, hacia una zona de código que ya fue ejecutada y que ahora volverá a ejecutarse. Las repeticiones terminan cuando la variable `num` supera el valor de 10. En dicho caso, la condición `num > 10` se hace verdadera y se ejecuta la instrucción `goto fin`, que manda el programa hasta la etiqueta `fin` para continuar ejecutando las instrucciones que se encuentran debajo; en este caso, un mensaje de despedida.

PROBLEMAS RESUELTOS

1. Raíces anidadas

Compruebe la aproximación mostrada abajo. Para ello debe calcular la expresión de la izquierda. Detenga la secuencia de operaciones luego de aplicar la raíz cuadrada por 20º vez.

$$\sqrt{2\sqrt{2\sqrt{2\sqrt{\dots}}}} = 2$$

Solución:

Se nota un patrón que se repite claramente en la expresión de la izquierda. Ello nos inspira a definir una variable x y actualizarla del siguiente modo:

$$\begin{aligned}x &= 1 \\x &= \sqrt{2} \\x &= \sqrt{2\sqrt{2}} \\x &= \sqrt{2\sqrt{2\sqrt{2}}}\end{aligned}$$

Con 20 actualizaciones similares habríamos alcanzado la estimación que se nos pide. La regla de actualización sería la siguiente: En cada iteración el valor de x debe ser actualizado con la raíz cuadrada de su doble, empezando con $x=1$. Esto se implementa fácilmente con:

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

int main(void) {
    float x = 1;
    for(int i=0; i<20; i++)
        x = sqrt(2*x);
    printf("%.7f\n", x);
}
```

raices.c

El resultado obtenido es 1.9999986. Desde la 8^{va} iteración x toma valores entre 1.99 y 2.0.

El prototipo de la función `sqrt` que se usa para obtener la raíz cuadrada de un número se encuentra en `math.h`. Si trabaja con un sistema GNU/Linux es necesario añadir la clausula `-lm` a la hora de compilar programas que incluyan `math.h`. O sea, la forma correcta de compilar el programa anterior sería: `gcc raices.c -lm`.

La clausula `-lm` es necesaria para decirle al enlazador (*linker*) que mezcle nuestro código con la librería matemática. A diferencia de lo que ocurre con otras librerías, el enlace de código no se realiza automáticamente con `math.h`.

2. Sumar y multiplicar es fácil

Escriba un programa que calcule la siguiente suma (con 1000 términos):

$$\frac{\pi^2}{6} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \frac{1}{5^2} + \dots$$

y el siguiente producto (con 10 términos)

$$\frac{1}{1^2} \cdot \frac{3}{2^2} \cdot \frac{5}{3^2} \cdot \frac{7}{4^2} \cdots \cdot \frac{2N-1}{N^2}$$

Solución:

Empecemos por la suma. Note que los términos siguen un patrón específico. El n -ésimo término es de la forma $1/n^2$. Dichos términos deben acumularse en una variable S mientras n varía desde 1 hasta 1000. Esto se puede implementar como:

```
float S = 0;
for(int n=1; n<=1000; n++)
    S += 1.0/(n*n);
```

Se debe inicializar explícitamente el sumador S en cero, pues, de lo contrario, este podría contener algún valor arbitrario que alteraría la suma. Asumir que las variables no asignadas tienen por defecto el valor cero es un error de programación muy frecuente.

Es obligatorio escribir el numerador de cada sumando como número decimal, 1.0 . Si lo escribiéramos como entero, 1 , la división $1 / (n*n)$ arrojaría cero para $n > 1$, pues ese es el cociente de la división entera de cualquier número entre otro mayor.

Tampoco está permitido omitir los paréntesis en el denominador. Si escribimos $1.0/n*n$, el computador operaría primero $1.0/n$ y el resultado obtenido sería luego multiplicado por n .

En cuanto a la multiplicatoria, cada término también sigue un patrón. El n -ésimo término es de la forma $(2n-1)/n^2$. Análogamente, se deben acumular las multiplicaciones en una variable M mientras n varía desde 1 hasta 10, lo que se logra así:

```
float M = 1;
for(int n=1; n<=10; n++)
    M *= (float)(2*n-1) / (n*n);
```

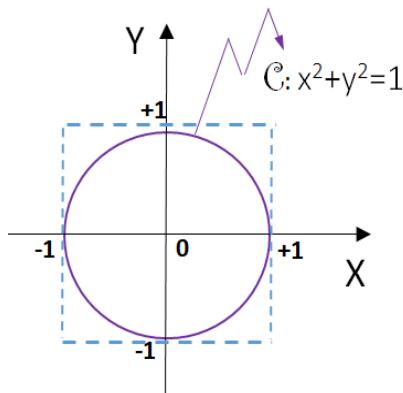
En este caso el valor de M debe inicializarse en 1 , el elemento neutro de la multiplicación.

Para evitar las divisiones enteras se tuvo que forzar el numerador, $2*n-1$, que es un número entero, para que sea tratado como flotante. Esto se logró usando conversión de tipos, (`float`). El mismo resultado se habría obtenido si en lugar de la conversión explícita se utilizaba $2*n-1.0$ como

numerador. En este caso el numerador ya no sería considerado como entero sino que sería convertido implícitamente al mismo tipo de `1.0`; es decir, a un número decimal.

3. Una ineficiente manera de aproximar π

Se sabe que si elegimos aleatoriamente un punto dentro del cuadrado punteado la probabilidad que dicho punto caiga dentro de la circunferencia inscrita es igual al área del círculo dividido entre el área del cuadrado. Usando esta idea es posible estimar el área del círculo unitario, cuyo valor es π , generando millones de puntos al azar dentro del cuadrado punteado y contando cuántos de estos caen dentro del círculo. Así, si el $p\%$ de los puntos generados aleatoriamente caen dentro del círculo, podríamos decir que el área del círculo es aproximadamente el $p\%$ del área del cuadrado. Se le pide estimar el valor de π usando la idea descrita.



Solución

El quid del asunto consiste en generar puntos (x, y) aleatorios dentro del cuadrado punteado. Para esto podríamos generar valores aleatorios e independientes de x en $[-1, 1]$ y de y en el mismo intervalo. Esto se logra usando la función `rand()` y la constante `RAND_MAX`, ambas definidas en el archivo `stdlib.h`.

La función `rand()` devuelve un número entero aleatorio que oscila entre 0 y `RAND_MAX`, siendo este último el valor de la constante `RAND_MAX`. Sabiendo esto, podemos generar números decimales aleatorios en el intervalo $[0, 1]$ simplemente haciendo:

```
(float) rand() /RAND_MAX.
```

El conversor de tipos `(float)` es para tratar el valor returnedo por `rand()` como si fuese un número flotante en lugar de un entero, evitando así que `rand() /RAND_MAX` sea tratada como una división entera, en cuyo caso siempre retornaría cero, como cada vez que se divide un número entero entre otro más grande. Como la expresión mostrada arriba genera un número en $[0, 1]$, al multiplicarla por dos obtendríamos un número en $[0, 2]$ y, si a esto le restamos una unidad, obtendríamos un número en el intervalo $[-1, 1]$, tal como estamos buscando. Así pues, la generación de un punto aleatorio dentro del cuadrado punteado se implementa con:

```
int x = 2*(float) rand() /RAND_MAX - 1;
int y = 2*(float) rand() /RAND_MAX - 1;
```

La expresión anterior deberá ejecutarse miles de veces, obviamente dentro un bucle. Cuantos más puntos aleatorios generemos más nos aproximaremos al valor real de π (aunque muy ineficientemente, debemos confesar).

Pero aún falta algo. Debemos contar cuántos de los miles de puntos (x, y) que vamos a «lanzar» dentro del cuadrado van a caer dentro del círculo. Esto es sencillo considerando que los puntos (x, y) que conforman el círculo unitario cumplen que: $x^2 + y^2 \leq 1$.

Abajo se muestra un programa completo que genera cien millones de puntos aleatorios.

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

int main(void) {
    float x, y;
    int cont=0;
    long N=100000000;

    for(int i=0; i<N; i++) {
        x = 2*(float)rand() / RAND_MAX - 1;
        y = 2*(float)rand() / RAND_MAX - 1;
        if(x*x + y*y <= 1)
            cont++;
    }

    float Area = 4.0 * cont/N;
    printf("pi=%f", Area);
}
```

La variable `cont` es un contador que se incrementa cada vez que un punto (x, y) cae dentro del círculo unitario; o sea, cuando se cumple $x^2 + y^2 \leq 1$.

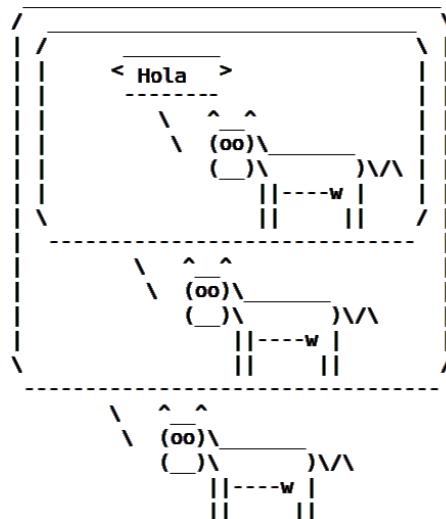
La expresión `cont/N`, casi al final del programa, representa la proporción entre la cantidad de puntos que cayeron dentro del círculo sobre los que cayeron dentro del cuadrado. Dicha proporción ha de ser aproximadamente la misma entre el área del círculo, π , y el área del cuadrado, $4 u^2$.

En nuestro caso, el programa arroja 3.141507 como estimación del área del círculo; o sea, como estimación de π . Existe una coincidencia de cuatro decimales con el valor real de π .

Finalmente, si ejecuta varias veces el programa mostrado notará que siempre va a obtener la misma respuesta. Esto se debe a que los valores generados por `rand` son pseudoaleatorios. La función `rand` siempre genera la misma secuencia de números. Para alterar este comportamiento simplemente añada el archivo `time.h` al inicio del programa y la instrucción `srand(time(NULL))` antes de iniciar el bucle. Esta última hará que, cada vez que se ejecute el programa, la función `rand()` generará secuencias aleatorias distintas, que dependerán de la hora del computador. Así se podrá obtener resultados distintos en cada ejecución.

4. La vaca loca

Implemente un programa que solicite al usuario un valor entero N e imprima una secuencia de imágenes como la mostrada abajo. La salida para $N=3$ sería la siguiente:



Solución:

La curiosa imagen de la vaquita parlante puede generarse con el programa `cowsay` que viene instalado en los sistemas GNU/Linux. Por ejemplo, la vaca más interna, la que dice «Hola», podría invocarse directamente desde la línea de comandos con `cowsay Hola`.

Se sabe también que GNU/Linux provee el operador `|` (*pipe*, en inglés), que sirve para enviar la salida de un programa como entrada de otro. Así, la sentencia `cowsay Hola | cowsay` envía la salida de `cowsay Hola` nuevamente al programa `cowsay`.

La salida de `cowsay Hola` es simplemente una secuencia de caracteres que en nuestra humana visión parece una respetuosa vaca. Al pasar esta salida a `cowsay` se dibujará una segunda vaca cuyo mensaje contiene todos los caracteres pasados a través del operador `|`; o sea, la imagen de la primera vaca diciendo «Hola». Así se forma la recursión.

Cuando el mensaje vacuno contiene muchos caracteres suele deformarse. Esta deformación se evita usando la cláusula `-n`, p.ej. `cowsay Hola | cowsay -n`.

Luego de esta introducción, la reproducción de N vacas en el formato pedido se lograría con:

```
cowsay Hola | cowsay -n | cowsay -n | cowsay -n | ... | cowsay -n
```

donde `cowsay` se repite N veces.

Este extenso comando debe ser construido en un programa C como el que se muestra a continuación:

```

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

int main (void) {
    char cmd[100];
    int N;

    puts("Ingrese un numero:");
    scanf("%d", &N);

    sprintf(cmd, "%s", "cowsay Hola");

    for(int i=1; i<N; i++)
        sprintf(cmd, "%s %s", cmd, "| cowsay -n");

    system(cmd);
}

```

La cadena de texto a construir se almacenará en `cmd`, que empieza con el valor `cowsay Hola`. Este valor inicial es asignado en la línea `sprintf(cmd, "%s", "cowsay Hola")`. Esta cadena inicial se concatenará con otras tantas subcadenas `"| cowsay -n"` como sean necesarias. Desgraciadamente, esto no será tan simple como hacer `cmd = cmd + "| cowsay -n"`. Esta forma sintáctica funciona bien para acumular números, pero no cadenas. En cambio, la instrucción:

```
sprintf(cmd, "%s %s", cmd, "| cowsay -n");
```

sí funciona correctamente con cadenas. Con esta expresión el valor asignado a `cmd` será la concatenación de dos cadenas, `"%s %s"`. La primera es el valor actual de `cmd`; y la segunda es la cadena `"| cowsay -n"`.

Cuando la instrucción anterior se repite `N` veces —dentro de `for(int i=1; i<N; i++)`— el valor final de `cmd` será exactamente la instrucción que queríamos construir.

Finalmente, para ejecutar la instrucción `cowsay Hola | cowsay -n | ... | cowsay -n`, ya construida y almacenada en `cmd`, se debe invocar la función `system` (en `stdlib.h`). Como ya se dijo antes, `system` permite ejecutar comandos de sistema desde un programa C.

El resto del código es trivial y el lector no merece ser aburrido con tales explicaciones.

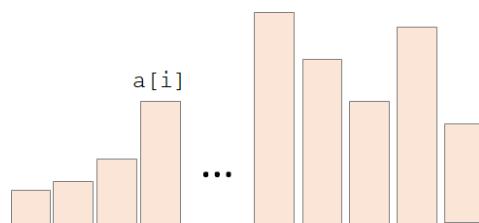
5. Ordenando un arreglo con el método de la burbuja

Escriba un programa que ordene ascendenteamente los valores de un arreglo de enteros.

Solución:

Existen varios métodos de ordenación. El más conocido utiliza el burbujeo como estrategia. Este método consta de tantos pasos como elementos contenga un arreglo a . La idea fundamental es la siguiente: En el i -ésimo paso de este proceso se debe analizar los elementos $a[i+1], a[i+2], a[i+3], \dots$ de modo que el menor de estos pase a ocupar posición de $a[i]$.

Para ilustrar lo anterior utilizaremos el siguiente gráfico. Aquí, los elementos del arreglo a se muestran como barras cuyos tamaños reflejan sus valores numéricos.



Justo en el momento mostrado ya se han ejecutado los tres primeros pasos de la ordenación y, por ello, las tres primeras posiciones ya contienen los menores valores de todo el arreglo. En dicho instante se está tratando de determinar qué elemento debe ir en la cuarta posición, $i=3$, y, para ello, se tiene que comparar $a[i]$ con todos los elementos que están a su derecha. Si se encuentra un elemento menor que $a[i]$ en la j -ésima posición ($j > i$), éste debe ser trasladado a la i -ésima posición. Para ello se deben intercambiar $a[i]$ y $a[j]$. Luego del intercambio se debe continuar comparando el nuevo $a[i]$ con los elementos restantes hasta llegar al final de la lista. Recién ahí concluye el i -ésimo paso del método. Todo lo dicho se implementa de la siguiente manera:

```
#include<stdio.h>
int main(void) {
    int a[N] = {6, 10, 4, 2, 8, 2, 11, 20, 5, 1};
    int N = sizeof(a)/sizeof(a[0]);
    int tmp;
    for(int i=0; i<N-1; i++) {
        for(int j=i+1; j<N; j++)
            if(a[j] < a[i]) {
                tmp = a[i];
                a[i] = a[j];
                a[j] = tmp;
            }
    }
}
```

El programa anterior ordena un arreglo de N elementos arbitrarios. Los $N-1$ pasos del método se ejecutan en el bucle externo, `for(int i=0; i<N-1; i++)`. En el i -ésimo paso se compara $a[i]$

con los elementos de su derecha ($j = i+1, i+2, \dots$), de modo que si alguno de estos elementos es menor (o sea si se cumple $a[j] < a[i]$), los valores de $a[i]$ y $a[j]$ son intercambiados. Las tres instrucciones dentro del bloque `if` permiten el intercambio de datos entre dos variables, $a[i]$ y $a[j]$, como ya se explicó en el primer ejercicio del capítulo anterior.

Sobra decir que con cambios menores el programa puede generalizarse para arreglos de cualquier tamaño y cualesquiera otros valores, que bien podrían ser ingresados desde el teclado.

6. Palíndromos

Un palíndromo es un texto que se lee igual de derecha a izquierda que de izquierda a derecha. Escriba un programa que solicite al usuario el ingreso de un texto y determine si este es palíndromo o no.

Solución:

El programa debe empezar pidiendo un texto y almacenándolo en un arreglo de caracteres.

```
char txt[100];
puts("Ingrese un texto:");
gets(txt);
```

La función `gets` (con prototipo en `stdio.h`) sirve para leer textos que contienen espacios en blanco. Si se usara `scanf("%s", txt)`, sólo se leería la primera palabra de tales textos.

El texto almacenado en `txt` es palíndromo si el primer carácter coincide con el último, el segundo con el penúltimo, el tercero con el antepenúltimo, y así sucesivamente. Esta idea se implementa así:

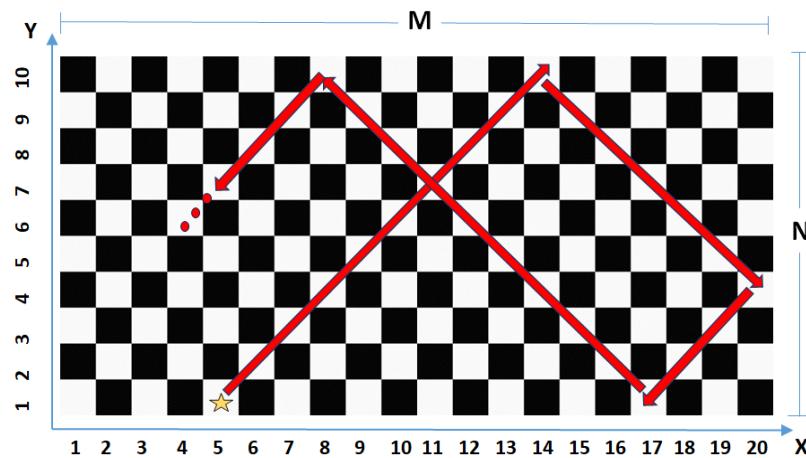
```
bool esPalindromo = true;
for(int i=0; i<strlen(txt)/2; i++)
    if(txt[i] != txt[strlen(txt)-1-i]) {
        esPalindromo = false;
        break;
    }
esPalindromo? puts("Sí es"): puts("No es");
```

La longitud del texto almacenado en `txt` se obtiene con `strlen(txt)`. Como ya se dijo, el i -ésimo carácter contado desde la izquierda, `txt[i]`, debe compararse con el i -ésimo carácter contado desde la derecha, `txt[strlen(txt)-1-i]`. Dado que ya no tendría sentido seguir haciendo comparaciones luego de haber alcanzado la mitad de la cadena, el bucle acabará cuando deje de cumplirse que $i < strlen(txt)/2$.

Justo antes de iniciar el bucle se empieza asumiendo que el texto a analizar es palíndromo, i.e. `esPalindromo = true`, pero apenas se encuentra que el i -ésimo carácter no coincide con su recíproco la asunción se descarta, i.e. `esPalindromo = false`. En ese momento se abandona el análisis, se sale del bucle usando `break`, y se pasa a mostrar el veredicto final al usuario.

7. Rebotando ando

En un tablero de N filas y M columnas como el mostrado abajo se coloca una bolita de goma en la posición $(x_{ini}, 1)$. Al ser lanzada en dirección noreste la bolita se moverá sobre ciertas casillas del tablero, rebotando cada vez que toca uno de los extremos del mismo, y dejará de rebotar cuando llegue de regreso a su casilla inicial. Se le pide escribir un programa que liste todas las casillas visitadas por la bolita en su travesía. Los valores de N , M y x_{ini} deben ser ingresados por el usuario.



Solución:

Para evitar implementar tediosas validaciones asumamos que el usuario ingresará valores sensatos; es decir $N > 1$, $M > 1$ y x_{ini} en $[1, M-1]$.

La solución propuesta consistirá de un bucle. Cada iteración del bucle actualizará la posición de la bolita, de su casilla actual a otra adyacente. El núcleo del programa se muestra abajo:

```

xincr = 1; yincr = 1;
x = xini;
y = 1;

do {
    x += incrX;
    y += incrY;
    printf("%d %d\n", x, y);
    if(x == 1 || x == N) incrX *= -1;
    if(y == 1 || y == M) incrY *= -1;
} while( !(x == xini && y == yini) );

```

Las variables x e y sirven para almacenar la posición actual de la bolita. Dichas variables serán impresas en cada iteración. Inicialmente haremos que $x=x_{ini}$ e $y=1$.

Definiremos otras dos variables, $incrX$ e $incrY$, para guardar la dirección en que se mueve la bolita. De este modo, si actualmente la bolita se encuentra en la posición (x, y) , en la siguiente iteración

deberá encontrarse en $(x+incr_x, y+incr_y)$. Dado que la bolita empieza moviéndose hacia el noreste, es decir, hacia la derecha y arriba, inicialmente se hace $incr_x=1$ e $incr_y=1$. Las variables $incr_x$ e $incr_y$ sólo pueden tomar $+1$ o -1 como valores. Note también que estos valores sólo cambian cuando la bolita choca con alguno de los extremos del tablero.

El programa se ejecuta siempre y cuando la bolita no regrese a su posición inicial; es decir, mientras no ocurra $x==x_{ini} \&& y==y_{ini}$.

En cuanto a los rebotes, debe observarse que el cambio de dirección depende de la pared con la que colisiona la bolita. Si esta choca con las paredes laterales ($x==1 \mid\mid x==N$), sólo cambiará la dirección horizontal del movimiento, no la vertical. Por ejemplo, si la bolita se movía hacia la derecha, $xincr=1$, luego del choque con una pared lateral se moverá hacia la izquierda, $xincr=-1$; independientemente de si venía subiendo ($yincr=1$) o bajando ($yincr=-1$).

Análogamente, si la bolita choca contra el piso o el techo ($y==1 \mid\mid y==M$), el choque sólo invertirá la variable $yincr$. Si antes subía, luego bajará; si antes bajaba, el choque la mandará de subida.

El cambio de dirección horizontal que sufrirá el movimiento se implementa como: $xincr*=-1$ (forma abreviada de $xincr=xincr*-1$). Análogamente, $yincr*=-1$ modifica la componente vertical del movimiento. Ambas asignaciones se limitan a cambiar el signo de sus respectivas variables de modo que estas siempre oscilen entre $+1$ y -1 .

La implementación mostrada también resuelve el espinoso caso cuando la bolita choca en una esquina. En dicha situación la bolita ha golpeado tanto una pared lateral como una vertical y por lo tanto se invertirán $xincr$ y $yincr$, con lo que la bolita regresará exactamente por donde vino.

Se le pide al lector que complete el programa añadiendo la solicitud de los valores N , M y x_{ini} al usuario. Eso es lo único que falta.

Finalmente, dejo una pregunta para el lector: ¿Será posible que para ciertos valores de N , M y x_{ini} la bolita se quede rebotando por siempre, sin volver nunca al punto inicial?

8. Matriz de Vandermonde

Escribir un programa que solicite al usuario un arreglo $\{a_0, a_1, \dots, a_{N-1}\}$ de tamaño N. El programa debe generar una matriz de la siguiente forma:

$$\left[\begin{array}{ccccc} 1 & a_0 & a_0^2 & \cdots & a_0^{N-1} \\ 1 & a_1 & a_1^2 & \cdots & a_1^{N-1} \\ 1 & a_2 & a_2^2 & \cdots & a_2^{N-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & a_{N-1} & a_{N-1}^2 & \cdots & a_{N-1}^{N-1} \end{array} \right]$$

Solución:

Antes de ejecutar el programa es imposible conocer de antemano cuantos datos querrá ingresar el usuario. Por ello debemos declarar un arreglo de longitud variable (VLA), cuyo tamaño N será definido por el usuario durante en tiempo de ejecución. El VLA servirá para almacenar los N valores que posteriormente ingresará el usuario. Note que N también influirá en la definición de la matriz solicitada, cuyas dimensiones, según el gráfico mostrado, serán ser $N \times N$.

Las definiciones de las estructuras requeridas podrían implementarse del siguiente modo:

```
int N;
puts("Ingrese tamaño del arreglo:");
scanf("%d", &N);
int coefs[N];
int Matriz[N][N];
```

Los datos del arreglo `coefs` deben ser ingresados, uno por uno, por el usuario:

```
for(int i=0; i<N; i++) {
    printf("Ingrese coefs[%d]:", i);
    scanf("%d", &coefs[i]);
}
```

La función `scanf` almacena el i-ésimo dato leído en la i-ésima posición del VLA, `&coefs[i]`.

Por otro lado, la matriz de Vandermonde debe ser generada a partir del arreglo `coefs`. Sabiendo que cualquier número elevado a la potencia cero es igual a uno, en general se cumple que el elemento `Matriz[i][j]`, con $i=0,1,\dots, N-1$ y $j=0, 1, \dots, N-1$ de la matriz de Vandermonde es $coefs[i]^j$. La exponenciación a la potencia j puede ser implementada con la función `pow` (del archivo de cabecera `math.h`), de la siguiente manera:

```

for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
        Matriz[i][j] = pow(coefs[i], j);

```

Luego de resolver las partes más críticas, se deja completar el programa al lector.

9. Midiendo la música

Defina una estructura Cancion que permita almacenar el título, cantante y duración (en minutos y segundos) de una canción. Luego utilice un arreglo de estas estructuras para almacenar una serie de canciones que serán ingresadas desde el teclado. El programa deberá calcular la duración total de todas las canciones de un cantante específico ingresado por el usuario.

Solución:

La estructura requerida para almacenar las canciones puede definirse así:

```

typedef struct song {
    char titulo[20];
    char cantor[20];
    int min;
    int seg;
} Cancion;

```

Todas las canciones ingresadas por el usuario podrían almacenarse dentro de un arreglo de canciones, playlist, cuyo contenido sería llenado interactivamente. Dicho proceso se implementa con un bucle que se repita tantas veces como canciones se desee registrar.

```

Cancion playlist[100];
int N;

puts("Ingrese número de canciones a registrar:");
scanf("%d", &N);
getc(stdin);

for(int i=0; i<N; i++) {
    puts("Ingrese titulo:");
    gets(playlist[i].titulo);

    puts("Ingrese cantante:");
    gets(playlist[i].cantor);

    puts("Ingrese duracion (mm:ss):");
    scanf("%d:%d", &playlist[i].min, &playlist[i].seg);
    getc(stdin);
}

```

Según el código anterior, el usuario ingresa el número de canciones que va a registrar, N.

Dentro del bucle, cada título ingresado se guarda en el atributo `titulo` del *i*-ésimo elemento del arreglo `playlist`, con `gets(playlist[i].titulo)`. Quienes trabajan en GNU/Linux, pueden evitar las advertencias de seguridad recibidas a causa del uso de `gets` usando en su reemplazo `fgets(playlist[i].titulo, 20, stdin)`.

Análogamente, el nombre de cada cantante se almacena en `playlist[i].cantor`.

El programa asume que la duración de una canción será ingresada como `mm:ss`, p.ej. `3:53`. Por ello, ambos datos pueden ser leídos con una sola instrucción `scanf` que este bien advertida del formato que usará el usuario, `%d:%d`. La duración de cada canción se graba en los atributos `min` y `seg` de `playlist[i]` usando:

```
scanf("%d:%d", &playlist[i].min, &playlist[i].seg)
```

No olvide de colocar el carácter ampersand, `&`, cada vez que lea datos numéricos con `scanf`; cuando se usa `gets` o `fgets` no se necesita dicho símbolo.

La instrucción `getc(stdin)` obedece a un detalle práctico importante. Úsela cada que vez que vaya a invocar la función `gets/fgets` luego de `scanf`. Esto evitará que el carácter de retorno (ENTER) sea consumido por `gets/fgets` evitando así un comportamiento indeseado.

Para sumar las duraciones de todas las canciones de un cantante específico se puede hacer:

```
char cantante[20];
puts("Ingrese el nombre de un cantante:");
gets(cantante);

int total_segs = 0;
for(int i=0; i<N; i++)
    if(strcmp(playlist[i].cantor, cantante) == 0)
        total_segs += 60*playlist[i].min + playlist[i].seg;
```

La variable `total_segs` acumula el número de segundos de todas las canciones de `cantante`. Note que la comparación de cadenas de texto requiere la función `strcmp` (con prototipo en `string.h`), que devuelve cero cuando las dos cadenas pasadas como argumentos son iguales en contenido. Sería un error intentar algo como `playlist[i].cantor==cantante`. Esto no sería una comparación de cadenas, sino una comparación de punteros.

Debe notarse que aunque las canciones suelen durar unos pocos minutos varias de ellas podrían durar horas en conjunto. Por lo tanto, aún falta traducir la duración total, `total_segs`, a su equivalente en horas, minutos y segundos. Para ello bastan unas pocas operaciones aritméticas:

```
int hh = total_segs / 3600;
int mm = (total_segs % 3600) / 60;
int ss = (total_segs % 3600) % 60;

printf("Las canciones de %s duran %02d:%02d:%02d", cantante, hh, mm, ss);
```

El total de horas, minutos y segundos contenidos en `total_segs` segundos se almacena en las variables `hh`, `mm` y `ss`, respectivamente.

El especificador de formato `%02d` obliga al sistema a mostrar los valores de `hh`, `mm` y `ss` con dos dígitos, completando con ceros a la izquierda cuando sea necesario.

10. En busca de Fischer, nuevamente

Una de las maneras de evaluar la situación de cada jugador en una partida de ajedrez consiste en sumar sus puntos. Cada pieza tiene un valor numérico que cuantifica su poderío. Así, la experiencia ajedrecística ha enseñado que un peón tiene un valor de un punto; el alfil vale 3 puntos; el caballo, 3; la torre, 5; y la dama, 9 puntos. Se le pide modificar el programa de la sección anterior para que pueda calcular el valor total de las piezas negras y de las blancas.



Solución:

Añadiremos dos bucles anidados al final del programa realizado el capítulo anterior:

```
#include <stdio.h>

typedef enum tipo_pieza {
    vacio=0, peon, torre, caballo, alfil, dama, rey
} Tipo;

typedef enum color_pieza {
    blanco, negro
} Color;

enum columnas {ca=0, cb, cc, cd, ce, cf, cg, ch};
enum filas {f8=0, f7, f6, f5, f4, f3, f2, f1};

typedef struct pieza {
    Tipo tp;
    Color cl;
} Pieza;

int main(void) {
    Pieza tablero[8][8] = {0};
    tablero[f2][cc] = (Pieza){alfil, blanco};
    tablero[f5][cc] = (Pieza){rey, negro};
    tablero[f6][cb] = (Pieza){peon, blanco};
    tablero[f7][cc] = (Pieza){rey, blanco};
    tablero[f3][cf] = (Pieza){alfil, negro};
```

```

// código para sumar puntos de cada jugador
int valor_pieza[] = {0, 1, 5, 3, 3, 9, 0};
int total_blancas = 0, total_negras = 0;
for(int i=0; i<8; i++)
    for(int j=0; j<8; j++)
        if(tablero[i][j].tp > 0)
            if(tablero[i][j].cl == blanco)
                total_blancas += valor_pieza[tablero[i][j].tp];
            else
                total_negras += valor_pieza[tablero[i][j].tp];

printf("Puntaje blanco =%d\n", total_blancas);
printf("Puntaje negro =%d\n", total_negras);
}

```

El arreglo `valor_pieza` contiene los valores de cada tipo de pieza. Los datos de este arreglo están alineados con los de la enumeración `enum tipo_pieza`. Esto se hace para que el valor de un alfil, por ejemplo, pueda ser obtenido directamente con `valor_pieza[alfil]`.

Las variables `total_blancas` y `total_negras` son sumadores. Sirven para acumular el total que puntos que posee cada jugador por sus piezas. Dichos valores se obtienen al visitar cada casilla del tablero y sumar el valor de la pieza ocupante al bando correcto.

Recuerde que las casillas del tablero, `tablero[i][j]`, no están representadas por número, caracteres ni ningún dato atómico, sino por una estructura de tipo `Pieza` con dos atributos: el tipo de la pieza contenida, `tp`, y el color de la misma, `cl`. En las casillas vacías ambos atributos son ceros debido a la asignación inicial `Pieza tablero[8][8] = {0}`.

Por lo anterior, la condición `tablero[i][j].tp > 0` sólo es verdadera cuando existe alguna pieza en `tablero[i][j]`. En este caso, el color de la pieza será `tablero[i][j].cl` y su valor puede ser obtenido a partir de su tipo, usando `valor_pieza[tablero[i][j].tp]`.

Nada malo ocurre si omitimos la condición `tablero[i][j].tp > 0`. En dichos casos, las casillas vacías no añadirían valor a las variables `total_blancas` y `total_negras` puesto que, como ya se dijo, `tablero[i][j].tp` sería cero y también lo es `valor_pieza[0]`. El único problema de omitir la condición citada sería que el programa realizaría más operaciones de las necesarias.

Al final, la salida del programa arroja un puntaje de 4 puntos (alfil + peón) para las piezas blancas y 3 (alfil) para las piezas negras.

Para quienes hayan quedado con la curiosidad insatisfecha aclaramos que el puntaje del rey siempre se ignora en la valoración de una partida. Dado que su presencia es obligatoria y sólo puede haber un rey por bando, esta pieza nunca es factor de desequilibrio material.

PROBLEMAS PROPUESTOS

1. Crear un programa que solicite al usuario los tres lados de un triángulo y determine si los datos ingresados son factibles. Se sabe que cada lado de un triángulo debe ser menor que la suma de los otros dos. Los valores a ingresar pueden ser números decimales.
2. Crear un programa que solicite al usuario los tres lados de un triángulo y determine si el triángulo es pitagórico; es decir, si se cumple que la longitud de alguno de los lados elevado al cuadrado es igual a la suma de cuadrados de los otros dos. Los valores a ingresar deben ser enteros.
3. Crear un programa que solicite el ingreso de un número racional, p.ej. 12/45, y que imprima la misma fracción factorizada, p.ej. 4/15.
4. Con la técnica que utilizamos en un ejercicio anterior para estimar el valor de π , estime el área encerrada entre la curva $100-x^2$, la recta $x=0$ y la recta $y=0$.
5. Una recta del plano XY puede ser definida con uno cualquiera de sus puntos, (x, y) , y su pendiente m . Implemente un programa que solicite dos rectas y determine el punto donde ambas se cruzan, o un mensaje que advierta que las rectas son paralelas.
6. Crear un programa que determine si los tres puntos (x, y) ingresados por el usuario son colineales.
7. Implementar un programa que transforme una cadena ingresada por el usuario, de modo que los caracteres iniciales de cada palabra sean cambiados a mayúsculas, y el resto a minúsculas. La función `tolower` (`toupper`), que retorna el carácter pasado como argumento en minúscula (mayúscula), puede ser invocada incluyendo el archivo `ctype.h`.
8. Crear un programa que detecte palíndromos ignorando los espacios en blanco. En este caso, la oración: «anita lava la tina» sería un palíndromo.
9. Escribir un programa que solicite al usuario el ingreso de una cadena y que elimine de esta los espacios blancos innecesarios. Por ejemplo, la cadena «anita lava la tina» debería convertirse en «anita lava la tina», con un único espacio blanco entre dos palabras.
10. Sean `paises` y `pobs` dos arreglos que contienen los nombres y las poblaciones (en millones) de N países, respectivamente. Se le pide crear un programa que elabore un reporte con los nombres y poblaciones de todos los países ordenados descendente por población.
11. Crear un programa que solicite N enteros al usuario. El programa deberá mostrar una única vez cada número ingresado; es decir, no se debe mostrar valores repetidos.

12. Crear un programa que solicite al usuario dos arreglos de enteros A y B y retorne la unión, la intersección y la diferencia de ambos conjuntos.
13. Crear un programa que solicite al usuario el ingreso de una matriz de dimensión $N \times M$ y que luego muestre su matriz transpuesta.
14. Crear un programa que solicite al usuario el ingreso de una matriz cuadrada y determine si es triangular superior o no.
15. **Crucigrama.** Crear una matriz de letras de 30×50 . Cada letra debe ser generada al azar de modo que nunca haya dos vocales adyacentes, ni horizontal, ni vertical, ni diagonalmente. Las letras desde la A hasta la Z tienen códigos ASCII en el rango de 65 a 90.
16. **Problema de las ocho reinas.** Mediante un programa determine cómo podrían colocarse 8 reinas sobre un tablero de ajedrez sin que ninguna de estas amenace a otra; es decir, sin que ningún par de reinas ocupe la misma fila, columna o diagonal. Si desea, puede complicar el problema y extenderlo a una versión de 1000 reinas sobre un tablero de 1000×1000 . De obtener la solución quizás aún pueda cobrar el millón de dólares que la universidad escocesa de Saint Andrews ofreció en el 2017 a quien pudiera resolver dicho acertijo.
17. Crear un programa que solicite al usuario un conjunto de puntos en el plano XY. Luego, esos mismos puntos deberán mostrarse en pantalla pero ordenados según su distancia al origen de coordenadas. Cada punto debe almacenarse en una estructura `Punto`, cuyos atributos `x` e `y` sean de tipo flotante.
18. Crear un programa que solicite un conjunto de números complejos y que calcule su producto. Cada número complejo debe almacenarse en una estructura `Complejo`, cuyos atributos sean los flotantes `real` e `imag`.
19. Crear un programa que solicite un conjunto de números racionales y que calcule su suma. Cada número racional debe almacenarse en una estructura `Racional`, cuyos atributos sean los enteros `numerador` y `denominador`.
20. **Matriz espiral.** Implementar un programa que imprima una matriz espiral de $N \times N$ elementos. La matriz contiene la secuencia de números desde 1 hasta N , que se enrrolla alrededor del centro. A continuación se muestra la matriz espiral para $N=5$.

1	2	3	4	5
16	17	18	19	6
15	24	25	20	7
14	23	22	21	8
13	12	11	10	9

21. **Matriz splash.** Implementar un programa que imprima una matriz de FILS x COLS elementos. En la posición (*i,j*), ingresada por usuario, la matriz debe contener un cero. Cada celda adyacente debe contener una unidad más, y así sucesivamente, tal como se muestra abajo:

```
$ ./a
Ingrese FILS:7
Ingrese COLS:12
Ingrese fila inicial:2
Ingrese columna inicial:3
3 2 2 2 2 2 3 4 5 6 7 8
3 2 1 1 1 2 3 4 5 6 7 8
3 2 1 0 1 2 3 4 5 6 7 8
3 2 1 1 1 2 3 4 5 6 7 8
3 2 2 2 2 2 3 4 5 6 7 8
3 3 3 3 3 3 3 4 5 6 7 8
4 4 4 4 4 4 4 5 6 7 8

$ ./a
Ingrese FILS:10
Ingrese COLS:7
Ingrese fila inicial:7
Ingrese columna inicial:5
7 7 7 7 7 7 7
6 6 6 6 6 6 6
5 5 5 5 5 5 5
5 4 4 4 4 4 4
5 4 3 3 3 3 3
5 4 3 2 2 2 2
5 4 3 2 1 1 1
5 4 3 2 1 1 1
5 4 3 2 2 2 2
```


CAPÍTULO

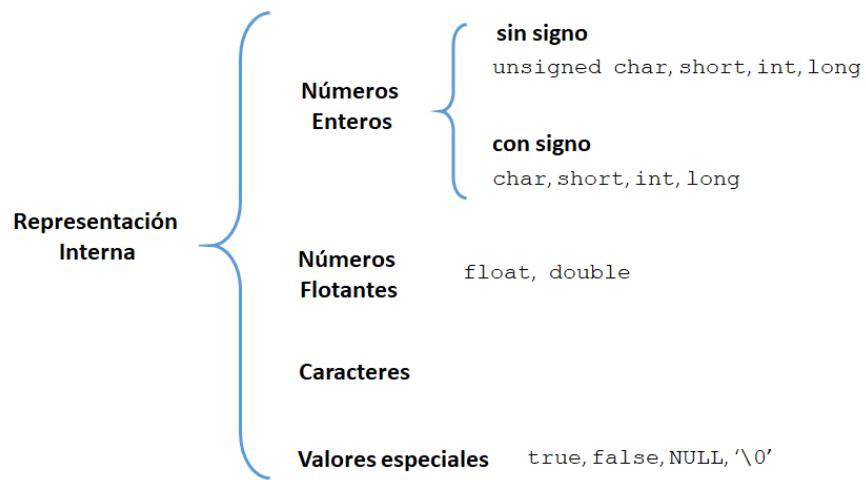
3

REPRESENTACIÓN INTERNA DE DATOS ATÓMICOS

Todos los datos que procesa un computador se almacenan en la memoria como secuencias de 0's y 1's. Las reglas que se utilizan para construir estas secuencias de 0's y 1's dependen del tipo de dato que se está representando, p.ej. el número 1 tiene distintas representaciones dependiendo de si se almacena como dato entero, `int`, o flotante, `float`.

En este capítulo estudiaremos cómo se representan internamente los datos; es decir, cómo el sistema codifica cada dato como cadenas de 0's y 1's dependiendo de su tipo. A través de una serie de ejemplos explicaré las operaciones manuales que debemos realizar para determinar la representación interna de un dato. Posteriormente, utilizando depuradores y programas propios, accederemos a la memoria para verificar que nuestros cálculos manuales realmente coinciden con el contenido físico de la memoria.

Las reglas de codificación que discutiremos se estudiarán separadamente, por tipos, siguiendo el siguiente esquema:



La representación interna de datos agregados (arreglos, cadenas, matrices, estructuras, etc.) se estudiará en un capítulo posterior.

REPRESENTACIÓN INTERNA DE ENTEROS

Los pasos a seguir para poder codificar un dato entero depende de si este ha sido declarado como alguno de los tipos signados (`char`, `short`, `int`, `long`), o bajo alguno de los tipos sin signo (`unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`).

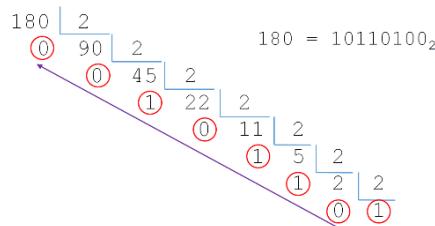
ENTEROS SIN SIGNO

El procedimiento para codificar un dato entero sin signo es bastante directo: Debe convertir el entero a base binaria y, posteriormente, agregarle tantos ceros a la izquierda como se requiera, hasta llenar el espacio de memoria reservado para dicho número.

Ejemplo 1. ¿Cómo se representa el número 180 cuando es de tipo `unsigned short`?

Solución:

Para determinar cómo sería la representación interna de 180 debemos obtener primero su representación binaria. Para ello debemos realizar las siguientes divisiones sucesivas:



Ahora que ya sabemos que el número 180 se representa en binario como la cadena de bits `10110100` sólo nos falta añadir unos cuantos ceros a la izquierda de esta cadena, hasta completar dos bytes, pues este es el espacio que ocupan los datos de tipo `unsigned short` en memoria.

La representación interna de 180 sería la siguiente:

00000000	10110100
----------	----------

Normalmente mostraremos la representación interna de un dato en formato hexadecimal, por ser éste un formato más legible, mucho más fácil de leer que una larga secuencia de bits.

La cadena de bits anterior en formato hexadecimal sería:

00	B4
----	----

En los gráficos mostrados, tanto en la representación binaria como en la hexadecimal, cada rectángulo contiene 8 bits y, por lo tanto, representa una celda de la memoria.

Recordatorio: Para convertir una cadena de bits a su representación hexadecimal existe un método directo que consiste en reemplazar cada bloque de cuatro bits (empezando por los 4 bits menos significativos, los del extremo derecho) por un dígito hexadecimal, según las siguientes tablas:

binario	hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

binario	hexadecimal
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Ejemplo 2. ¿Cómo se representaría el número 180 si fuera de tipo `unsigned int`?

Solución:

Lo único que cambia con respecto al ejemplo anterior es que ahora el dato 180 será representado con 4 bytes, pues ese es el espacio de memoria reservado para los datos de tipo `unsigned int`. La representación binaria de 180 es obviamente la misma que la calculada anteriormente; su representación interna es también muy similar sólo que ahora posee más ceros en la parte izquierda.

00000000	00000000	00000000	10110100
----------	----------	----------	----------

En formato hexadecimal sería así:

00	00	00	B4
----	----	----	----

ENTEROS CON SIGNO

Cuando trabajamos con datos de tipo `char`, `short`, `int` ó `long`, las reglas de codificación que debemos aplicar dependen de si el dato a guardar es positivo o negativo.

Si el dato es un entero positivo, los pasos a seguir para obtener su representación interna son los mismos que los descritos anteriormente. Si es un entero negativo, se debe aplicar una operación adicional conocida como **complemento a dos**, que denominaremos como C-2.

Ejemplo 1. ¿Cómo se representa el dato 180 cuando es de tipo `long`?

Solución:

Como el dato a codificar es positivo, el procedimiento es el mismo que el seguido en la subsección anterior. Intuyendo la impaciencia de mis lectores pasaré directamente a mostrar el resultado:

00000000	00000000	00000000	00000000	00000000	00000000	00000000	10110100
----------	----------	----------	----------	----------	----------	----------	----------

Ejemplo 2. ¿Cómo se representa el dato -180 al ser asignado a una variable de tipo `short`?**Solución:**

Ahora sí el procedimiento es distinto. Para hallar la representación de -180 debemos partir de la representación de su valor absoluto, 180, que, según un ejemplo anterior, es la siguiente:

00000000	10110100
----------	----------

Ahora debemos aplicar el complemento a dos, C-2, a esta secuencia de bits. Esta operación se realiza en dos pasos: Primero se invierten todos los bits mostrados: los ceros se vuelven unos y viceversa. Luego, se debe sumar una unidad a la cadena de bits invertida.

Según lo dicho, el C-2(180) se obtiene de la siguiente manera:

00000000	10110100
----------	----------

↓
Paso 1: Inversión de bits

11111111	01001011
----------	----------

+
↓
Paso 2: Sumar 1 bit
1

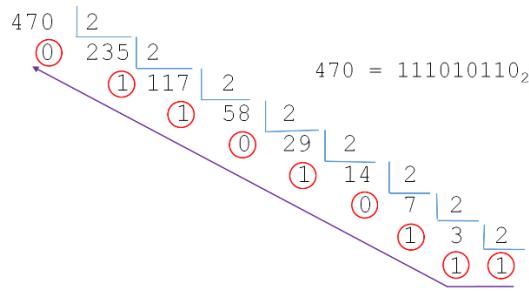
11111111	01001100
----------	----------

La cadena 11111111 01001100 es la representación interna de -180 (`short`). Una manera equivalente pero más legible de mostrar la representación interna es la siguiente:

FF	4C
----	----

Ejemplo 3. ¿Cómo se almacena internamente -470 de tipo `short`?**Solución:**

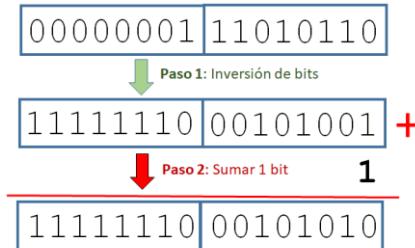
Debemos hallar la representación de 470 (positivo) y obtener su complemento a dos. Haciendo divisiones sucesivas entre dos se observa cómo sería 470 en base binaria.



Dado que dicho valor será almacenado en una variable de tipo `short`, la representación obtenida debe ser completada con ceros a la izquierda hasta llenar dos bytes. Así se obtiene el siguiente resultado intermedio:

00000001	11010110
----------	----------

Esto último es la representación del entero positivo 470. Aún nos falta calcular el complemento a dos de esta secuencia de bits. Dicho resultado sería la representación interna de -470 y se obtiene de la siguiente manera:



En formato hexadecimal el número -470 se representaría así:

FE	2A
----	----

Observaciones

- Cuando trabajamos con enteros signados los números negativos siempre tendrán un 1 en el bit más significativo (el de más a la izquierda) de la representación interna; los positivos siempre tendrán 0 en el mismo bit. Dicho bit se conoce como el **bit de signo**. Este bit no contribuye a representar la magnitud de un número, i.e. su valor absoluto.
- Se cumple que: $C-2(C-2(x)) = x$

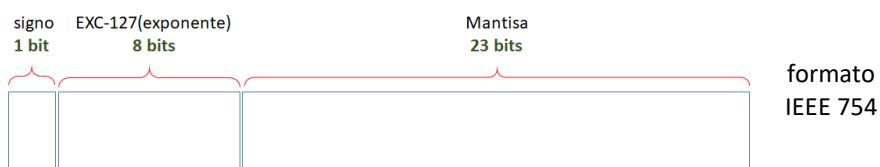
REPRESENTACIÓN INTERNA DE NÚMEROS DECIMALES

A diferencia de los tipos de datos estudiados anteriormente, los datos de punto flotante (`float`, `double`, `long double`) permiten almacenar números con parte decimal.

La representación interna de un número decimal posee tres partes: (1) el signo, (2) el exponente, y (3) la mantisa. En términos simples, la mantisa almacena los dígitos significativos del número; el exponente indica la posición del punto decimal; y el signo indica si el número es positivo o negativo.

NÚMEROS DE COMA FLOTANTE DE PRECISIÓN SIMPLE, `float`

Todo dato almacenado en una variable de tipo `float` ocupa cuatro bytes, que están organizados según el siguiente formato:



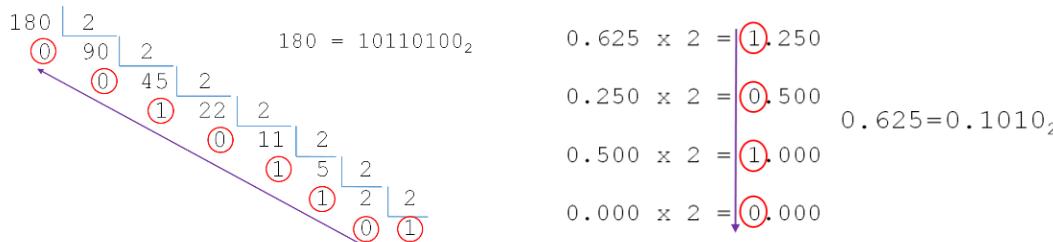
Los siguientes ejemplos ilustran cómo obtener cada una de estas partes.

Ejemplo 1: ¿Cómo se almacena internamente el valor -180.625 de tipo `float`?

Solución:

Empecemos por lo más fácil: el bit de signo. Dado que el número a almacenar es negativo, su representación interna incluirá un 1 en el primer bit, el bit de signo. Un cero en dicho bit significaría que el número que se está codificando es positivo, que no es el caso aquí.

Ahora pasemos a determinar cuáles serían los 23 bits que han componer la mantisa. Para ello necesitamos obtener la representación binaria de 180.625 (ignorando el signo negativo). Esto se consigue trabajando por separado la parte entera y decimal del número. Para convertir la parte entera a binario se hacen divisiones sucesivas mientras que la parte decimal se transforma haciendo multiplicaciones sucesivas.



De lo anterior se observa que 180.625 se representa en base binaria como 10110100.101_2 . Note que el último cero podría omitirse ya que los ceros de la parte decimal, a la derecha del último bit 1, no son significativos.

Para encontrar la mantisa necesitamos escribir el número binario recientemente obtenido en notación científica; es decir, en el siguiente formato:

$$1.b_1b_2b_3b_4 \dots \times 2^{\text{exponente}}$$

Cualquier número binario puede escribirse en este formato. Sólo debe moverse el punto binario algunos espacios a la derecha o a la izquierda, según sea necesario. En nuestro caso:

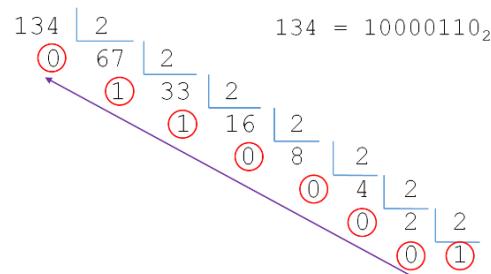
$$10110100.101_2 = 1.0110100101_2 \times 2^7$$

La operación anterior no es más que una analogía de las operaciones que solemos realizar mentalmente en nuestra acostumbrada base decimal, donde sabemos que desplazar el punto decimal tres espacios a la derecha de un número y luego dividirlo entre 1000, o correr el punto decimal dos espacios a la izquierda y multiplicar por 100, por ejemplo, no afecta el valor de un número.

De la representación anterior ya podemos obtener la mantisa. Es la secuencia de bits que sigue al punto binario, o sea 0110100101 . Es cierto que aquí no hay 23 bits y por ello debemos agregar varios 0's hasta completar 23 bits. Pero, a diferencia de lo que hacíamos con los números enteros, esta vez no añadiremos los 0's a la izquierda, sino a la derecha de la cadena obtenida.

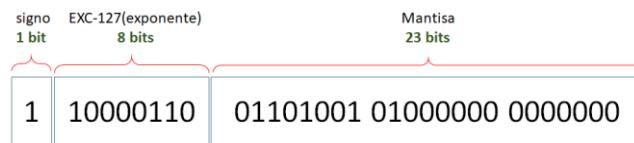
Para obtener el exponente aún falta realizar una operación adicional. El exponente obtenido en la notación científica, o sea el 7, debe ser añadido al número 127. El resultado, 134, debe ser transformado a binario y recién allí obtendríamos los 8 bits que conforman la zona denominada exponente según el formato IEEE 754. Si al convertir su resultado a binario se obtuviera una cadena de más de 8 bits, esto significaría que ha tratado de guardar un valor fuera de rango en el tipo `float`.

La representación binaria de 134 sería la siguiente:



La operación de añadir 127 unidades al exponente de la notación científica se denomina **exceso a 127**, y se denota EXC-127.

Habiendo obtenido las tres partes sólo falta juntarlas. El número -180.625 del tipo float tendrá la siguiente representación interna:



En formato hexadecimal sería:

C3 34 A0 00

Ejemplo 2: ¿Cómo se almacena internamente el valor flotante 0.7?

Solución:

Sin necesidad de cálculo podemos afirmar que el bit de signo en la representación de 0.7 será el 0, puesto que 0.7 es un número positivo.

Luego, para obtener la mantisa y el exponente debemos transformar el número 0.7 a binario.

$$\begin{aligned}
 0.7 \times 2 &= 1.4 \\
 0.4 \times 2 &= 0.8 \\
 0.8 \times 2 &= 1.6 \\
 0.6 \times 2 &= 1.2 \\
 0.2 \times 2 &= 0.4 \\
 0.4 \times 2 &= 0.8 \\
 0.8 \times 2 &= 1.6 \\
 0.6 \times 2 &= 1.2 \\
 &\vdots
 \end{aligned}$$

Vemos que algo curioso ha ocurrido. Podríamos seguir multiplicando indefinidamente y las multiplicaciones nunca convergirán en una secuencia de ceros, como ocurrió en el ejemplo anterior.

Pero, felizmente, existe un patrón que se repite. Siendo así, la representación binaria de 0.7 será una representación periódica mixta, tal como se ve a continuación:

$$0.7 = 0.1011001100110\dots_2 = 0.\overline{10110}_2$$

Sin dejarnos intimidar por los infinitos paquetes de 0110's que contiene el número obtenido, podemos expresarlo en notación científica de la siguiente manera:

$$0.1011001100110\dots_2 = 1.\overline{0110}_2 \times 2^{-1}$$

Ahora vemos que este problema tiene dos ingredientes adicionales con respecto al anterior. Primero, ahora tenemos una mantisa de infinitos bits; segundo, ahora tenemos un exponente negativo.

En cuanto a la mantisa, resulta obvio que no podremos almacenar sus infinitos bits. En la memoria sólo hay espacio para los primeros 23 bits. ¿Qué pasará con el resto de bits?, ¿serán simplemente descartados sin pena ni gloria? La respuesta dependerá del 24º bit, el primer bit a descartar. Si este fuese cero, entonces la mantisa quedaría conformada por los primeros y afortunados 23 bits que lograron alcanzar un espacio en la memoria; el resto de bits sería descartado, así sin más. Pero si el 24º bit fuese uno, entonces deberíamos añadirle un bit a la mantisa. Más directamente, cuando la mantisa tiene más de 23 de bits se debe redondear (en binario) a 23 bits.

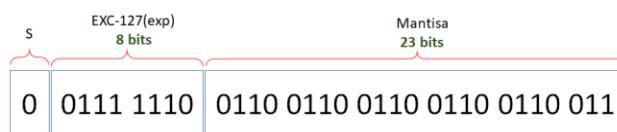
En particular, para este ejemplo, el primer bit de la cadena a descartar sería cero. Esto se deduce desarrollando el periodo 0110, tal como se muestra abajo.



Luego, los 23 bits que lograron alcanzar un espacio en la zona reservada para la mantisa se quedan tal como están; no hay necesidad de añadir ningún bit a la mantisa.

Finalmente, tenemos que añadir 127 al exponente obtenido en la notación científica, o sea a -1. El resultado, 126, se representa como 111 1110 en base binaria. Cuando, como en este caso, el exponente a exceso 127 tiene menos de 8 bits, se le debe agregar ceros a la izquierda.

La representación del número 0.7 de tipo `float` sería así:

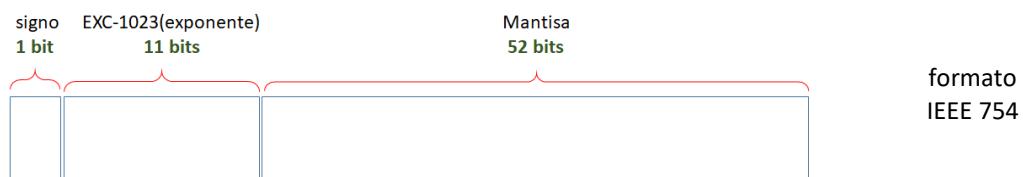


Y en hexadecimal:

3F 33 33 33

NÚMEROS DE COMA FLOTANTE DE PRECISIÓN DOBLE, `double`

Los números decimales almacenados en una variable de tipo `double` tienen una representación interna con el siguiente formato:



Al igual que con los decimales de precisión simple, aquí también se observan las mismas tres componentes: signo, exponente y mantisa.

El proceso a seguir para determinar la representación interna de un `double` es similar al estudiado en la sección anterior. Los datos deben convertirse a binario y expresarse en notación científica. De allí se obtiene la mantisa, que ahora debe ser representada con 52 bits. El exponente obtenido en la notación científica debe ser añadido en 1023, EXC-1023, para recién conocer los 11 bits del exponente. El bit de signo será 0 si el número a representar es positivo; y 1 en caso contrario.

El tipo `double` permite una precisión de 15 a 17 dígitos decimales, aproximadamente el doble de precisión que el tipo `float`, que permite guardar de 6 a 9 dígitos decimales significativos.

Ejemplo 1. ¿Cómo se representaría el dato $1+2^{-52}$ de tipo `double`?

Solución:

No hay necesidad de resolver la expresión $1+2^{-52}$ para luego convertir el resultado a binario. Las potencias de dos (positivas o negativas) pueden ser convertidas directamente a binario sin tener que hacer muchas operaciones.

De las siguientes equivalencias: $2^{-1} = 0.1_2$, $2^{-2} = 0.01_2$, $2^{-3} = 0.001_2$, se puede deducir que el número 2^{-52} en binario sería $0.000\dots0001_2$ (con 51 ceros entre el punto binario y el 1). Por lo tanto, el número $1+2^{-52}$, en notación científica, sería:

$$1.000\dots0001_2 \times 2^0, \text{ con 51 ceros entre los dos } 1's$$

El bit de signo sería 0, dado que se trata de un número positivo. La mantisa sería la cadena de 52 bits que sigue al punto binario, o sea $000\dots0001$, que cabe exactamente en los 52 bits que permite del formato IEEE754. El exponente a exceso 1023 sería $1023+0$. Para ahorrarnos las divisiones sucesivas basta recordar que $2^{2-1} = 11_2$, $2^{3-1} = 111_2$, $2^{4-1} = 1111_2$,... para inferir que $1023=2^{10}-1$ tendría como representación binaria la cadena $11\ 1111\ 1111$.

Juntando las partes ya identificadas, la representación interna de $1+2^{-52}$ como tipo `double` sería:

0	011 1111 1111	0000 0000 0000 0000 ... 0000 0000 0000 0001
---	---------------	---

También podríamos expresar lo anterior más compactamente:

3F	F0	00	00	00	00	00	01
----	----	----	----	----	----	----	----

De todos los números mayores que la unidad, $1+2^{-52}$ es el número más pequeño que puede almacenarse con exactitud (sin tener que truncar la mantisa) en un dato de tipo `double`.

REPRESENTACIÓN INTERNA DE CARACTERES Y VALORES ESPECIALES

CARACTERES

La mayoría de los caracteres que uno utiliza con frecuencia (letras mayúsculas, letras minúsculas, dígitos, signos de puntuación, etc.) pueden ser almacenados en una variable de tipo `char`. En este caso, la representación del carácter almacenado coincide con la de su código ASCII. Por ejemplo, los caracteres '`'A'`', '`'@'`', '`'\n'`', cuyos códigos ASCII son 65, 64 y 13, respectivamente, tendrán como representación interna las cadenas `01000001`, `01000000` y `00001101`, respectivamente; o sea, las representaciones binarias de 65, 64 y 13.

Pero también existe un conjunto de caracteres que poseen códigos ASCII extendidos (mayores que 127), p.ej. `Ñ`, `á`, `ë`, `®, ü`, etc. Estos no pueden almacenarse dentro de una variable de tipo `char`. Los caracteres con código ASCII extendido deben ser almacenados en dos bytes, para cual deben ser definidos como caracteres largos, `wchar_t`, tal como se muestra abajo:

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

int main(void) {
    wchar_t var = L'Ñ';
    setlocale(LC_ALL, "");
    printf("%lc : %d", var, var);
}
```

El código extendido de `Ñ` es 241 (`00H D1H`) y, por lo tanto, su representación interna es la misma que la de 241, `00000000 11010001`.

El tipo `wchar_t` no es un tipo predefinido. Para usarlo debe incluirse el archivo `wchar.h`.

VALORES ESPECIALES

- Es posible declarar variables booleanas utilizando el tipo `bool`, definido en el archivo `stdbool.h`. Estas variables, que se almacenan en un único byte, sólo pueden aceptar dos valores, `true` o `false`. El valor `true` se representa internamente como la cadena de bits `00000001`; y el valor `false` como `00000000`.
- Es una práctica común inicializar un puntero con el valor `NULL` mientras se espera el momento apropiado para asignarle una dirección específica. Mientras un puntero está asignado con el valor `NULL`, sus 64 bits (8 bytes) están apagados. El valor `NULL` se almacena en memoria como:

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

- El carácter de terminación '`\0`' sirve para indicar el final de una cadena de caracteres. Su representación interna es `00000000`.
- Cuando se trabaja con datos flotantes, `float`, el valor representado puede caer en una las siguientes tres categorías, dependiendo de los 8 bits de exponente:
 - **Caso 1:** Cuando todos los bits del exponente son 1's, entonces:
 - Si la mantisa está llena de 0's, el número es `inf` (infinito positivo) ó `-inf` (infinito negativo), dependiendo del bit de signo. Estos valores se generan, por ejemplo, en divisiones donde el divisor es muy próximo a cero. Existen funciones como `isinf` (`math.h`) que permiten manipular estos números.
 - Si la mantisa no está llena de ceros, el número representado es `nan` ó `-nan`, un valor extraño (*not a number*) que depende del bit de signo. Estos valores aparecen, por ejemplo, al tratar de obtener la raíz cuadrada de un negativo. Existen funciones como `isnan` (`math.h`) que permiten manipular estos números.
 - **Caso 2:** Cuando todos los bits del exponente son 0's, entonces el valor representado se considera un **número denormalizado** (también llamado subnormal). Se trata de números pequeños, cercanos a cero, que son codificados con una regla distinta a la usada para números normales. La decodificación de un número denormalizado se hace mediante la siguiente fórmula:

$$(-1)^{\text{signo}} \times (0.\text{mantisa}) \times 2^{-126}$$

- **Caso 3:** En el resto de casos, cuando el exponente oscila entre `01H` y `FEH`, el valor representado es un **número normalizado**. Estos se decodifican mediante la fórmula:

$$(-1)^{\text{signo}} \times (1.\text{mantisa}) \times 2^{\text{exp}-127}$$

donde `exp` es el valor (decimal) codificado en los 8 bits del exponente.

- Cuando se trabaja con datos flotantes, `float`, existen dos ceros, positivo y negativo.
 - El cero positivo es una secuencia de 4 bytes llenos de 0's. Este aparece, por ejemplo, en divisiones donde el divisor es mucho mayor y tiene el mismo signo que el dividendo.
 - El cero negativo tiene 1 en el bit de signo y 0's en los otros 31 bits. Este aparece, por ejemplo, en divisiones donde el divisor es mucho mayor y de signo opuesto al dividendo.
- Los valores especiales discutidos para el tipo `float` también existen para el tipo `double`. Las representaciones internas de estos últimos son análogas a las que acabamos de comentar.

ENDIANNES

El *endianness* es una característica del sistema que determina cómo deben ser almacenados físicamente los datos que ocupan más de un byte.

Las secuencias de bits que obtuvimos mediante cálculos manuales en las secciones anteriores no necesariamente reflejan la representación física de los datos, su disposición real en la memoria; sólo eran una representación abstracta. Antes dijimos, por ejemplo, que el entero `-470` de tipo `short` se representa internamente como `FE 2A`. Esto no significa que si pudiéramos entrar a la memoria del computador al momento de la ejecución y ver los 2 bytes reservados para la variable que contiene el dato `-470` encontraríamos necesariamente la secuencia de bits `FE 2A`. Esto sólo ocurriría en uno de dos posibles casos: cuando la computadora que ejecutó la asignación, `short y = -470`, posee una **arquitectura *big-endian***. Pero existe una segunda posibilidad: que dicha computadora posea una **arquitectura *little-endian***. En este último caso todavía debemos hacer algunas aclaraciones.

Una computadora sólo puede poseer una de dos arquitecturas: *little-endian* o *big-endian*. Ninguna es mejor que la otra. Simplemente cada una almacena los datos en memoria de manera distinta. Utilizando un depurador o implementando un programa es posible determinar el *endianness* de nuestro sistema. El primero de estos métodos será utilizado al final de la sección.

BIG ENDIAN y LITTLE ENDIAN

La representación física de un dato almacenado en una arquitectura *big-endian* coincide exactamente con su representación abstracta, la cual se obtiene siguiendo las reglas descritas en las secciones previas. En cambio, la representación física en una *little-endian* es similar a la de una *big-endian*, ambas utilizan los mismos bytes, pero dispuestos en orden inverso.

Usando los ejemplos anteriores pasaremos a ver las diferencias de representación de un mismo dato en ambos tipos de arquitectura.

El entero `unsigned short var = 180` se almacenaría físicamente de la siguiente forma:

En *big-endian*

00	B4
----	----

En *little-endian*

B4	00
----	----

El entero `unsigned int var = 180` se guardaría, dependiendo de la arquitectura, así:

En *big-endian*

00	00	00	B4
----	----	----	----

En *little-endian*

B4	00	00	00
----	----	----	----

El entero negativo `short var = -180` se almacenaría en memoria así:

En *big-endian*

FF	4C
----	----

En *little-endian*

4C	FF
----	----

El entero negativo `short var = -470` se representaría como:

En *big-endian*

FE	2A
----	----

En *little-endian*

2A	FE
----	----

El número flotante `float var = -180.625` se grabaría, dependiendo de la arquitectura, así:

En *big-endian*

C3	34	A0	00
----	----	----	----

En *little-endian*

00	A0	34	C3
----	----	----	----

El número flotante `float var = 0.7` se almacenaría en memoria, dependiendo de la arquitectura, así:

En *big-endian*

3F	33	33	33
----	----	----	----

En *little-endian*

33	33	33	3F
----	----	----	----

El número double `x = 1 + pow(2, -52)` se representaría, dependiendo de la arquitectura, así:

En *big-endian*

3F	F0	00	00	00	00	00	01
----	----	----	----	----	----	----	----

En *little-endian*

01	00	00	00	00	00	F0	3F
----	----	----	----	----	----	----	----

El carácter `char var = 'A'` se guardaría de la siguiente manera:

En *big-endian*

41

En *little-endian*

41

El carácter largo `char var = 'Ñ'` se guardaría de la siguiente manera:

En *big-endian*

00 D1

En *little-endian*

D1 00

Los ejemplos anteriores son elocuentes. En resumen, las primeras secciones de este capítulo nos enseñaron a calcular cómo se representan los datos en una arquitectura *big-endian*. Si la arquitectura hubiese sido *little-endian* la representación hubiera sido similar, excepto que invertida a nivel de bytes, no a nivel de bits, como muchos suelen confundir.

Note que los datos que ocupan un byte (p.ej. `char`) se representan de la misma manera en cualquier arquitectura. Sólo los datos que ocupan más de un byte tienen una representación dependiente del *endianness* del computador.

VERIFICANDO CON `gdb`

Para verificar lo que se ha expuesto utilizaremos el depurador `gdb` (GNU debugger). Por decirlo de una manera simple, un **depurador** es una herramienta que nos permite ejecutar programas en cámara lenta, permitiéndonos decidir manualmente en qué momento ejecutar cada instrucción. De esta forma tenemos tiempo suficiente para inspeccionar las variables de un programa durante su ejecución, normalmente con la finalidad de encontrar errores en el código.

A continuación mostraremos cómo usar `gdb` para observar la representación física de un dato almacenado en una variable `var`, definida como `unsigned int var = 180`. Para ello deben seguirse los siguientes pasos:

- **Paso 1:** Escribir el escueto programa mostrado abajo en el archivo `prog.c`

```
int main(void) {
    unsigned int var = 180;
}
```

- **Paso 2:** Compilarlo con la cláusula `-g`; es decir, ejecutar: `gcc -g prog.c`
- **Paso 3:** Depurar el ejecutable obtenido haciendo: `gdb a.out` (ó `gdb a.exe` si trabaja en Windows). Esto hará que pase del entorno bash al entorno `gdb`. El prompt cambiará a (`(gdb)`).
- **Paso 4:** Dentro del entorno de trabajo de `gdb` deberá iniciar la ejecución en modo de depuración con el comando `start`.

- Paso 5: Se debe indicar al depurador que ejecute la primera línea de código (i.e. la declaración y asignación de la variable `var`). Para ello utilice el comando `next`. En modo de depuración, el programa siempre se queda en «Pausa» después que se ejecuta una instrucción.

```
$ gdb a ← Paso 3
GNU gdb (GDB) (Cygwin 10.2-1) 10.2
Copyright (c) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-cygwin".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"
...
Reading symbols from a...
(gdb) start ← Paso 4
Temporary breakpoint 1 at 0x10040108d: file ejm.c, line 5.
Starting program: /home/User/a
[New Thread 11864.0x2328]

Thread 1 "a" hit Temporary breakpoint 1, main () at ejm.c:5
5      unsigned int var = 180;
(gdb) next ← Paso 5
6  }
(gdb)
```

- Paso 6: Después de ejecutar los comandos anteriores la variable `var` ya está cargada en memoria con el valor 180. Aprovechando que el programa está en modo «Pausa», ese es el momento adecuado para:
 - Averiguar la dirección de `var`, lo que se logra escribiendo el comando `p &var` y
 - Echar un vistazo a los cuatro bytes que se encuentran a partir de esta dirección. Esto se logra con el comando `x/4bx` seguido de la dirección obtenida en el paso anterior.

En el gráfico mostrado abajo, los cuatro bytes de la última línea, `b4 00 00 00`, serían la representación física de 180, el contenido de las cuatro celdas de memoria que se encuentran a

partir de `0xfffffcc1c`. El prefijo `0x` sólo es un indicativo de que se está mostrando el contenido de cada celda en formato hexadecimal. Comparando la salida mostrada por `gdb` con los resultados manuales que calculamos anteriormente se deduce que mi computadora posee una arquitectura *little-endian*.

Paso 6.a
\$1 = (unsigned int *) 0xfffffcc1c
(gdb) x/4bx 0xfffffcc1c ← **Paso 6.b**
0xfffffcc1c: 0xb4 0x00 0x00 0x00
(gdb) |" data-bbox="173 196 842 593"/>

```

<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"
...
Reading symbols from a...
(gdb) start
Temporary breakpoint 1 at 0x10040108d: file ejm.c, line 5.
Starting program: /home/User/a
[New Thread 11864.0x2328]

Thread 1 "a" hit Temporary breakpoint 1, main () at ejm.c:5
5      unsigned int var = 180;
(gdb) next
6
(gdb) p & var ← Paso 6.a
$1 = (unsigned int *) 0xfffffcc1c
(gdb) x/4bx 0xfffffcc1c ← Paso 6.b
0xfffffcc1c:    0xb4    0x00    0x00    0x00
(gdb) |

```

Notas sobre gdb

- El comando `p` se usa para imprimir variables y/o direcciones de variables.
- El comando `x/4bx [dir]` permite examinar cuatro bytes (4b) en formato hexadecimal (`x`) a partir de una dirección `[dir]`. En el ejemplo mostrado el valor de `[dir]` fue `0xfffffcc1c`.
- Para salir de `gdb` debe usarse el comando `q` (del inglés `quit`)

PROBLEMAS RESUELTOS

1. **Números inexactos.** Verifique que el flotante 0.1 no puede ser almacenado con exactitud.

Solución:

Transformando 0.1 a base 2 nos encontramos con la siguiente secuencia:

$$\begin{aligned}0.1 \times 2 &= 0.2 \\0.2 \times 2 &= 0.4 \\0.4 \times 2 &= 0.8 \\0.8 \times 2 &= 1.6 \\0.6 \times 2 &= 1.2 \\0.2 \times 2 &= 0.4 \\&\vdots\end{aligned}$$

Así pues tenemos que:

$$0.1 = 0.00011001100110011\dots = 0.\overline{0001}_2$$

Lo que en notación científica sería:

$$0.1 = 1.1001100110011_2 \times 2^{-4} = 1.\overline{1001}_2 \times 2^{-4}$$

De lo anterior se observa que la mantisa sería una secuencia de infinitos cuartetos de bits 1001; el exponente sería la representación binaria de $\text{EXC-127}(-4) = 123$, o sea, 0111 1011. El bit de signo sería cero ya que estamos trabajando con un número positivo.

Juntando las tres partes del formato IEEE754 obtenemos:



Note que el primero de los infinitos bits que serán perdidos a causa de las limitaciones de espacio es uno, 1. Debido a esto todavía nos falta añadir una unidad a la mantisa. Esto dejaría la representación interna (en *big-endian*) del siguiente modo:



o, equivalentemente:

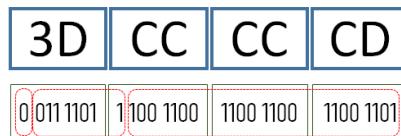


Debe darse cuenta que esta no es la representación exacta de 0.1, es sólo una aproximación. La representación exacta de 0.1 posee una mantisa de infinitos bits, la cual, por limitaciones de espacio, no se puede almacenar completamente. Utilizar una variable de tipo `double` no resolvería el problema, aunque sí se podría guardar más bits que con el tipo `float`. La representación obtenida usando `double` sólo sería más precisa, pero jamás exacta.

- 2. Decodificando el formato IEEE-754.** Si no es 0.1, ¿cuál es el número que está representado exactamente detrás de la secuencia 3D CC CC CD en una arquitectura *big-endian*?

Solución:

A continuación mostramos el número que se desea decodificar en bits.



Del gráfico anterior se observa que la primera parte del formato IEEE-754, el bit signo, es 0; la segunda parte, EXC-127(exp), es 0111 1011 (123, en decimal); y la tercera parte, la mantisa, es la cadena de bits 100 1100 1100 1100 1100 1101.

Invertiendo el proceso utilizado para construir la representación interna de un número es posible deducir el número detrás de 3D CC CC CD. Para esto empezamos con la siguiente fórmula:

$$(-1)^{\text{signo}} \times (1.\text{mantisa}) \times 2^{\text{exp}-127}$$

El flotante que se encuentra codificado detrás de la cadena mostrada sería:

$$(-1)^0 \times (1.10011001100110011001101) \times 2^{123-127}$$

23 bits

Para obtener la notación científica del número buscado debemos desplazar el punto binario cuatro espacios a la izquierda. Esto a causa de la potencia $2^{123-127} = 2^{-4}$. El número buscado, expresado base binaria, sería:

$$0.000110011001100110011001101_2$$

Convirtiéndolo a base decimal tendríamos:

$$2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + 2^{-16} + 2^{-17} + 2^{-20} + 2^{-21} + 2^{-23}$$

Lo anterior resulta en 0.1000002384185791015625 , un número un poco mayor (pero diferente) que 0.1 .

- 3. Next.** Determine cuál es el siguiente entero mayor a 50,000,000 que puede representarse exactamente en una variable de tipo `float`.

Solución:

Adelantamos que la respuesta no es 50 millones y una unidad, como uno podría estar tentado a pensar. Esto sería cierto en una variable entera, pero no en una flotante.

Usando una calculadora podemos encontrar la representación hexadecimal de 50 millones y escribirlo en binario:

$$2 \text{ FA F0 80}_H = 10\ 1111010\ 1110000\ 1000000_2$$

Este número en notación científica sería:

$$1.0\ 1111010\ 1110000\ 1000000_2 \times 2^{25}$$

De lo anterior se puede deducir que la representación de este número tendría como exponente a $\text{EXC-127}(25) = 152 = 1001\ 1000_2$; como bit de signo al cero; y como mantisa a los primeros 23 bits de la cadena $0\ 1111010\ 1110000\ 1000000$. La representación interna de 50 millones como dato flotante sería:

0	1001 1000	011 1110 1011 1100 0010 0000
---	-----------	------------------------------

A partir de esta representación recién podemos deducir el siguiente número flotante que se puede representar exactamente (sin pérdida de bits) en formato IEEE-754. Es casi el mismo número, excepto que con un bit adicional en la mantisa. O sea:

0	1001 1000	011 1110 1011 1100 0010 0001
---	-----------	------------------------------

Usando la fórmula de decodificación del ejercicio anterior, el número representado en la cadena de bits mostrada sería:

$$1.011110101110000100001_2 \times 2^{152-127}$$

Resolviendo se obtiene:

$$101111010111000010000100_2 = 50000004$$

De lo anterior se puede deducir que a partir de cierto umbral el rango de los números flotantes deja de incluir a todos los números enteros. Números como 50000003, por ejemplo, no pueden ser representados con exactitud en formato IEEE-754.

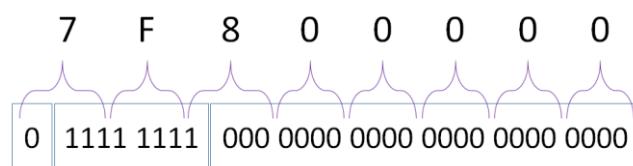
- 4. Tocando el infinito.** Al utilizar `gdb` para ver el contenido de los cuatro bytes reservados para una variable flotante se observó lo siguiente:

7F	80	00	00
----	----	----	----

¿Cuál es el número representado por dicha cadena en una arquitectura *big-endian*?

Solución:

Como ya vimos en ejercicios anteriores, para decodificar un flotante codificado en formato IEEE-754 se necesita conocer cada una de sus tres partes: signo, exponente y mantisa. Por ello mostramos los 4 bytes bajo análisis en base binaria:



Ahora ya se puede identificar fácilmente el signo (0), el exponente (1111 1111) y la mantisa (una lista de 0's) del dato flotante cuyo valor deseamos deducir.

En este caso no será necesario realizar operaciones de decodificación. Ya habíamos dicho que cuando el exponente del formato IEEE754 está llenos de 1's el número representado tiene un significado especial. En este caso se trata del infinito positivo, `inf`. También pudo haber sido un valor no numérico, `nan`, si alguno de los 23 bits de la mantisa hubiese sido 1.

Así pues, la salida del programa mostrado será `inf`.

El valor `inf` puede manipularse con funciones como `isinf`, que se encuentran dentro del archivo `math.h`.

PROBLEMAS PROPUESTOS

1. Determine cómo se almacena el número `-1` cuando es asignado a una variable `int` y cuando se asigna a una variable `float`.
2. Deduzca el mayor valor positivo y el menor valor negativo que puede ser almacenado dentro de una variable `char`. Recuerde que `char` es un tipo signado y su primer bit está reservado para representar el signo del dato almacenado.
3. Las variables

```
signed char x = -64;
unsigned char y = 192;
```

poseen la misma representación interna. Adicionalmente, el valor absoluto del dato sin signo, 192, es el triple del dato signado, 64. Encuentre otra pareja de valores para `x` y `y`, que posean la misma representación interna, pero donde el dato sin signo sea siete veces el dato signado.

4. Suponga que se ha definido `unsigned char var = 129`. ¿Qué valor obtendríamos si intentásemos decodificar dicho dato como si fuera de tipo `char`?
5. Una unión es un tipo de datos que permite que varias variables comparten un mismo espacio de memoria. Con esta ventaja un mismo dato puede ser interpretado como si fuera de distintos tipos. Por ejemplo, en el código mostrado, el número `0x3d800000` es grabado como entero, pero impreso como si fuera flotante. ¿Cuál será la salida del programa?

```
#include <stdio.h>

union number {
    int ivalue;
    float fvalue;
};

int main(void) {
    union number n1;
    n1.ivalue = 0x3d800000;
    printf("%6f", n1.fvalue);
}
```

6. Determine cuál es el número positivo más pequeño que puede almacenarse con exactitud dentro de una variable de tipo `float`.
7. Pruebe que el número más próximo y mayor a `1984` que puede grabarse exactamente en una variable de tipo `float` es `1984.0001220703125`.
8. Encuentre el primer entero `N` tal que, al ser almacenado como número flotante, tiene una representación interna inexacta.

CAPÍTULO

4

PUNTEROS A DATOS ATÓMICOS

Un puntero es una variable cuyo valor es una dirección de memoria. Esta dirección suele ser la dirección de otra variable, pero también podría ser la dirección de un arreglo, de una estructura o de una función previamente definida.

El buen uso de punteros nos permitirá, entre otras cosas, pasar funciones como argumentos a otras funciones, reservar y liberar manualmente ciertas zonas de memoria y crear estructuras dinámicas de datos, como pilas, colas y árboles, entre otros.

DECLARACIÓN Y ASIGNACIÓN DE PUNTEROS

Para declarar un puntero debe colocarse un asterisco entre el nombre del puntero y el tipo de datos al que este ha de apuntar, por ejemplo:

```
int * ptr;
```

Según la declaración anterior, el puntero `ptr` puede y debe almacenar la dirección de una variable entera. Para asignarle un valor a `ptr` se utiliza la siguiente notación:

```
ptr = &var;
```

El símbolo `&` (se lee ampersand) se llama **operador de dirección**. Se coloca a la izquierda de `var` para obtener la dirección de esta variable, la cual será asignada a `ptr`.

La instrucción anterior será correcta siempre que la variable `var` haya sido previamente declarada como variable entera (`int var`). Si `var` fuese de un tipo distinto de `int` la asignación anterior provocaría que el compilador arroje una advertencia de tipos de punteros incompatibles.

Luego de la asignación `ptr = &var` se puede decir que `ptr` apunta a la variable `var`. Para verificar que esta asignación ha sido exitosa y que ahora `ptr` contiene la dirección de `var` basta con imprimir ambos valores y comprobar que son iguales.

```
printf("La dirección de var es %p", &var);  
printf("El valor contenido en ptr es %p", ptr);
```

El especificador de formato `%p` se usa cada vez que se desea imprimir una dirección de memoria.

También era posible realizar la declaración y asignación del puntero `ptr` en una misma línea haciendo:

```
int * ptr = &var;
```

En general, la sintaxis

<code>[tipo de dato] * [nombre puntero] = [dirección];</code>

es válida para declarar punteros que apuntan a tipos simples, predefinidos, como `char`, `short`, `double`, por citar algunos ejemplos. La declaración de punteros a estructuras de datos o funciones es más intrincada y será estudiada en capítulos posteriores.

TAMAÑO DE UN PUNTERO

Cuando se trabaja sobre una arquitectura de 64 bits el espacio que ocupan los punteros en la memoria es 8 bytes. Esto siempre es cierto sin importar el tipo del puntero, el cual podría apuntar a una variable entera, a un carácter o a una función muy compleja. En arquitecturas de 32 bits el tamaño de los punteros es de 4 bytes.

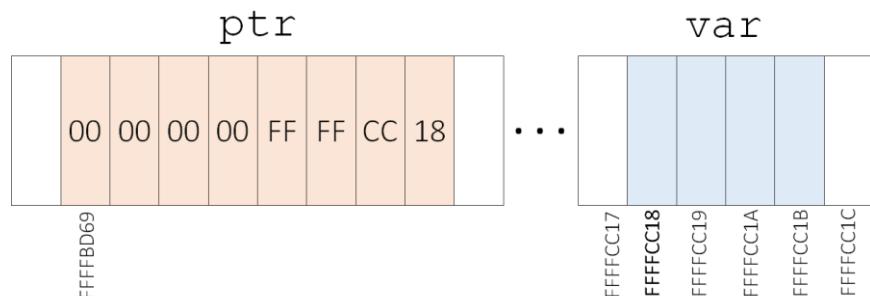
Para conocer el tamaño de un puntero podemos utilizar la función `sizeof` de la siguiente forma:

```
printf("El tamaño de ptr es %lu", sizeof(ptr));
```

El especificador `%lu` (`long unsigned`) se usa para capturar el valor returned por `sizeof`. También puede utilizar el especificador `%d` si trabaja en Windows.

PUNTEROS EN MEMORIA

Si pudiéramos ver la memoria justo después de que la asignación `ptr = &var` ya haya sido ejecutada, podríamos encontrarnos con algo similar a esto:



Este gráfico muestra la memoria como una serie de celdas. Cada celda posee una dirección única y puede almacenar 8 bits (0's y 1's) de información. Note que las direcciones de las celdas aumentan de uno en uno, de izquierda a derecha, según el gráfico.

La variable `var` por ser del tipo `int` ocupará 4 celdas de memoria. Según el gráfico, estas celdas comienzan en `FFFFCC18` y terminan en `FFFFCC1B`. Dentro de estas celdas se almacena el valor asignado a `var`, obviamente codificado en 0's y 1's.

La variable `ptr`, por ser puntero, ocupará 8 celdas de memoria. En esos 8 bytes se guardará una dirección de memoria que, en nuestro caso, será la dirección de la primera de las cuatro celdas que fueron reservadas para `var`.

Durante la ejecución del programa los contenidos de `var` y `ptr` podrían cambiar, pero el espacio ocupado por estas variables será siempre el mismo. El contenido de las celdas reservadas para `var` cambiará cada vez que se asigne un nuevo valor a esta variable. El contenido de las celdas reservadas para `ptr` cambiará cada vez que se haga apuntar `ptr` hacia otra variable.

Note que el valor contenido en un puntero no es lo mismo que la dirección del puntero. En el gráfico, el valor de `ptr` es `FFFFCC18` (la dirección de `var`), mientras que la dirección de `ptr` es `FFFFBD69`. Ambas direcciones podrían ser impresas con el siguiente comando:

```
printf("ptr está almacenado en %p y apunta a %p", &ptr, ptr);
```

DESREFERENCIA DE UN PUNTERO

A través de un puntero uno puede acceder a la variable apuntada, ya sea para asignarle un nuevo valor o para leer su valor vigente. Ambas situaciones se aprecian en el siguiente ejemplo:

```
int var = 10
int *ptr = &var;
printf("%d", *ptr);    // lee e imprime el valor de var, 10
*ptr = 20;             // actualiza el valor de var a 20
```

En las dos primeras líneas del código anterior se define (1) una variable entera, `var`, y (2) un puntero a dicha variable, `ptr`. Las dos últimas líneas imprimen y actualizan el valor de `var`, respectivamente, sin que se haya tenido que hacer una referencia directa a esta variable, es decir, sin siquiera mencionarla. Tanto en su impresión como en su actualización nos referimos a `*ptr` en lugar de `var`.

Cada vez que colocamos un asterisco, `*`, delante del nombre de un puntero estamos desreferenciando el puntero. En este caso, el asterisco, `*`, se denomina **operador de desreferencia**. Mientras `ptr` se refiere al puntero en sí, `*ptr` se refiere al dato apuntado, el dato que se encuentra en la dirección contenida en `ptr`.

Por lo expuesto anteriormente, la instrucción `printf("%d", *ptr)` podría leerse como sigue: «Imprime el dato que se encuentra en la dirección `ptr`», mientras que `*ptr = 20` podría leerse así: «Actualiza el dato almacenado en la dirección `ptr` con el valor de 20».

El asterisco utilizado en la declaración de un puntero no debe ser interpretado como operador de desreferencia. En una sentencia como `int *p1, *p2`, por ejemplo, el asterisco sólo está indicando que `p1` y `p2` son punteros, pero no se está desreferenciando nada en dicho momento.

Es un error común intentar desreferenciar un puntero que no ha sido inicializado (i.e. que ha sido declarado, pero al que no se le ha asignado ningún valor aún). Por lo general esto conduce a resultados inesperados. Un patrón de código que podría utilizarse para evitar estos problemas consiste en inicializar a `NULL` cada puntero para luego verificar su nulidad, justo antes de que sea desreferenciado.

El siguiente esquema ilustra lo dicho:

```
int *ptr = NULL;      // Aún no se conoce el valor de ptr
...                  // Cualquier código aquí
assert(ptr!=NULL);
*ptr = 10;
```

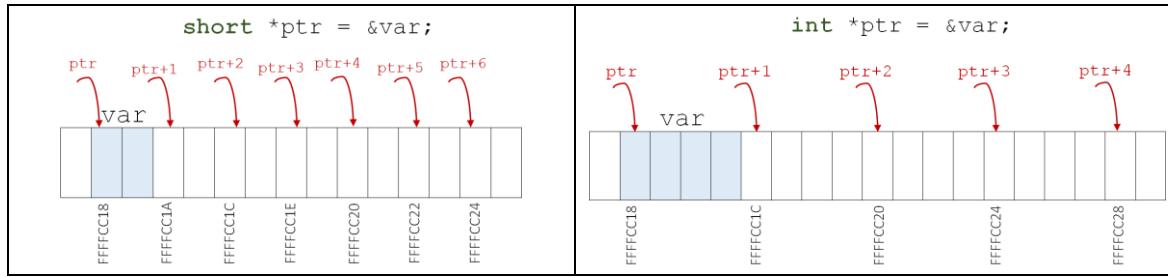
La función `assert` (del archivo `assert.h`) terminará el programa con un detallado mensaje que indica el lugar exacto del código donde se violó la asunción `ptr!=NULL`. El lector podría preferir utilizar condicional `if` en lugar de `assert` para manejar manualmente el error.

ARITMÉTICA DE PUNTEROS

Existen tres operaciones aritméticas que pueden realizarse con los punteros:

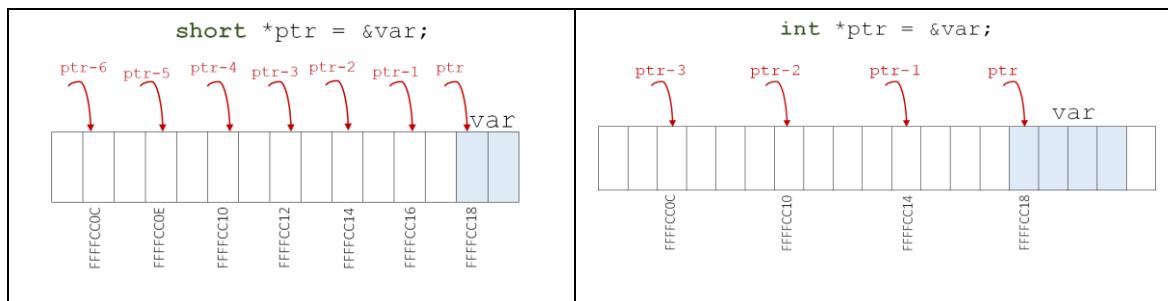
- Sumar un entero a un puntero, `ptr + k`
- Restar un entero de un puntero, `ptr - k`
- Restar dos punteros, `ptr1 - ptr2`

En cuanto a la suma, debe considerarse que no se trata de la suma aritmética que todos aprendemos en la escuela. La expresión `ptr + 1` hace referencia a una dirección de memoria, pero no necesariamente a la dirección de la siguiente celda a la que apunta `ptr`, como uno podría pensar. La dirección `ptr + 1` depende del tipo de datos al que apunta `ptr`. Si `ptr` apunta a una variable de tipo `short`, `ptr + 1` se refiere a la dirección de la celda que se encuentra 2 bytes después de `ptr`; si `ptr` apunta a una variable de tipo `int`, `ptr + 1` es la dirección que se encuentra 4 celdas después de `ptr`; y, como seguramente ya debe inferir, si `ptr` apunta al tipo `long`, `ptr + 1` apuntaría a la 8^{va} celda de memoria que está después de `ptr`. El gráfico mostrado abajo es elocuente.



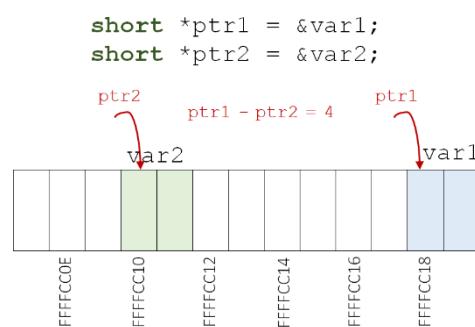
Sumar un entero a un puntero. La dirección `ptr+k` se encuentra a $k \cdot sz$ bytes de distancia a la derecha de `ptr`, donde `sz` es el tamaño (en bytes) de la variable apuntada por `ptr`.

La resta tiene una interpretación análoga. Así como `ptr+k` es una dirección que está $k \cdot sz$ bytes a la derecha de `ptr`, `ptr-k` se encuentra $k \cdot sz$ bytes a la izquierda de `ptr`. En ambos casos, `sz` se refiere al tamaño (en bytes) de la variable apuntada por `ptr`, o sea `sz = sizeof(*ptr)`.



Restar un entero de un puntero. La dirección `ptr-k` se encuentra a $k \cdot sz$ bytes de distancia a la izquierda de `ptr`, donde `sz` es el tamaño (en bytes) de la variable apuntada por `ptr`.

Finalmente, la resta de dos punteros, `ptr1 - ptr2`, sólo puede efectuarse si ambos son del mismo tipo. A diferencia de las operaciones anteriores, el resultado de la resta de punteros es un número entero, no una dirección. Este número es igual a la cantidad de elementos que caben entre `ptr2` y `ptr1`. Nos referimos a elementos del mismo tipo que los datos apuntados por `ptr1` y `ptr2`.



Restar dos punteros. `ptr1 - ptr2` indica cuántos elementos de tipo short cabrían entre estas dos direcciones.

PUNTEROS Y ARREGLOS

Aprovechando el hecho de que todos los elementos de un arreglo son del mismo tipo y ocupan posiciones adyacentes en la memoria, es posible acceder a cualquier elemento de un arreglo a partir de un puntero a su primer elemento.

Si asignamos la dirección del primer elemento de un arreglo al puntero `ptr`, entonces la expresión `ptr+1` se referirá a la dirección del segundo elemento; `ptr+2`, a la del tercero, y así sucesivamente. Al desreferenciar las direcciones `*ptr`, `*(ptr+1)`, `*(ptr+2)`,..., lograríamos acceder a todos los elementos contenidos en el arreglo. Un bloque de código muy comúnmente utilizado para navegar e imprimir los elementos de un arreglo tiene la siguiente forma:

```
int array[SIZE];
...
int *ptr=&array[0];           // dirección del primer elemento de array
for(int k=0; k<SIZE; k++)
    printf("%d", *(ptr+k)); // imprime el k-ésimo elemento de array
```

OPERADORES UNARIOS DE INCREMENTO Y DECREMENTO

También es posible utilizar los operadores unarios de incremento, `++`, y decremento, `--`, sobre punteros. Pero hay que tener en cuenta que estos cambiarán el valor del puntero.

El siguiente programa imprimirá el segundo elemento del arreglo `array`, pero `ptr` se mantendrá apuntando al primer elemento.

```
int* ptr = &array[0];
printf("%d", *(ptr+1));
```

En cambio, el siguiente programa también imprimirá el segundo elemento de `array`, pero además cambiará el valor de `ptr`, que terminará apuntado al elemento impreso.

```
int* ptr = &array[0];
++ptr;
printf("%d", *ptr);
```

Esto se debe a que la línea `++ptr` equivale a `ptr=ptr+1`. Aquí es donde se actualiza el valor del puntero `ptr`. La misma discusión se aplica para el operador de decremento.

DECAIMIENTO DE UN ARREGLO

Los arreglos decaen a punteros. Esta es una frase que no debe olvidar. Significa que el nombre de un arreglo casi siempre será interpretado como la dirección de su primer elemento. Por ello, los dos siguientes bloques de código son equivalentes:

<pre>int array[SIZE]; int *ptr = &array[0];</pre>	<pre>int array[SIZE]; int *ptr = array;</pre>
En ambos casos, <code>ptr</code> contiene la dirección del primer elemento de <code>array</code>	

En el primer bloque se indica explícitamente que `ptr` debe contener la dirección del primer elemento de `array`. En el segundo bloque se aprovecha el hecho que los arreglos decaen y, por ello, la asignación sólo utiliza el nombre `array`, a sabiendas que este nombre será interpretado como la dirección de su primer elemento, `&array[0]`.

Excepciones

De lo dicho anteriormente el lector podría creer que las expresiones `&array[0]` y `array` son intercambiables. No siempre es así. Existen tres excepciones, tres situaciones en las que el nombre del arreglo no será interpretado como una dirección, sino como todo el arreglo en sí.

- El primer caso excepcional ocurre cuando se utiliza el operador `sizeof`. La expresión `sizeof(array)` retornará el espacio ocupado por todo el arreglo. Si `array` estuviera compuesto por 10 elementos de tipo `int`, `sizeof(array)` devolvería $10 \times 4 = 40$. Esto es muy diferente que usar `sizeof(&array[0])`, que devolverá 8, puesto que todas las direcciones de memoria y, en particular, la dirección de `array[0]`, ocupan 8 bytes.
- La segunda excepción ocurre con el operador unario de dirección `&`. La expresión `&array` se interpreta como la dirección del arreglo `array`, no como la dirección de la dirección del primer elemento, `&&array[0]`. De hecho, esta última expresión ni siquiera tiene sentido.
- El tercer caso ocurre con el operador `typeof`. Asumiendo que existe un arreglo `int array[4]`, por ejemplo, la declaración `typeof(array) var` haría que `var` sea definida como un arreglo; en cambio, la declaración `typeof(&array[0]) var` la definiría como un puntero a entero.

SUBÍNDICES Y DESPLAZAMIENTOS

Uno puede acceder al $k+1$ -ésimo elemento de un arreglo, `array`, de dos formas: a través de la expresión `array[k]` o usando `*(array+k)`. En el primer caso se dice que estamos utilizando una **notación basada en subíndices** (*subscript notation* en inglés); en el segundo, una **notación basada en desplazamientos** (*offset notation* en inglés).

Para descifrar la expresión `*(array+k)` deben combinarse tres conceptos ya conocidos: desreferencia, aritmética de punteros y decaimiento. A modo de repaso diremos que `array` es la dirección del primer elemento del arreglo (decaimiento); `array+k` sería entonces la dirección del $k+1$ -ésimo elemento o, si se quiere, la dirección que se encuentra $k \times 4$ bytes a la derecha de `array` (aritmética de punteros); y, finalmente, `*(array+k)` es el valor que se encuentra almacenado en la dirección `array+k` (desreferencia).

Además del típico recorrido de un arreglo utilizando subíndices también existe el recorrido utilizando desplazamientos. El código tendría la siguiente forma:

```
int array[SIZE];
for(int k=0; k<SIZE; k++)
    printf("%d", *(array+k)); // imprime el k-ésimo elemento de array
```

De la equivalencia entre la notación por subíndices y la notación por desplazamientos se puede deducir:

Propiedad:

```
array[k] = *(array + k)
```

PUNTEROS GENÉRICOS (`void*`)

Existe un puntero que puede almacenar direcciones de cualquier tipo, p.ej. la dirección de una variable entera, de una variable flotante o incluso la dirección de una matriz o una función. Estamos hablando de un **puntero genérico**, que se declara de la siguiente manera:

```
void *ptr;
```

Estos punteros tienen dos propiedades: Primero, como ya se dijo, un puntero genérico puede almacenar cualquier dirección, sin importar el tipo del dato almacenado en dicha dirección. Por ello, son válidas las siguientes instrucciones:

```
int a; float b; char c;
void *ptr;
ptr = &a; // ptr apunta a una variable entera,
ptr = &b; // luego a una variable flotante,
ptr = &c; // y finalmente a una variable char
```

Segundo, un puntero genérico puede ser asignado a cualquier tipo de puntero. Suena similar a la propiedad anterior pero no es lo mismo. Esta propiedad hace que sean válidas las siguientes sentencias:

```
void *ptr = &var;
int *p1 = ptr;
char *p3 = ptr;
float *p2 = ptr;
```

Una desventaja de los punteros genéricos es que no pueden ser desreferenciados. Por eso mismo, el siguiente código arrojaría un error en tiempo de compilación en la última línea.

```
int var = 1984;
void* ptr = &var;
printf("%d", *ptr);
```

Esto se debe a que el sistema no puede recuperar el valor almacenado en `ptr` sin saber qué tipo de dato está almacenado en esa dirección. Si se supiera, por ejemplo, que en `ptr` está almacenado un dato de tipo `int`, entonces el sistema podría decodificar el valor almacenado en las 4 celdas que se encuentran a partir de la dirección `ptr`. Hablamos de 4 celdas porque ese es el espacio que ocupa un dato de tipo `int`. Lamentablemente, cuando declaramos `ptr` como puntero genérico no se conoce el tipo del dato apuntado y, por ende, es imposible saber cuántas celdas deben ser decodificadas para recuperar dicho dato.

La aritmética de los punteros genéricos está bien definida, aunque de manera arbitraria. La expresión `ptr+1` se refiere a la dirección de la celda inmediatamente posterior a la apuntada por `ptr`. O sea, `ptr+1` está un byte a la derecha de `ptr`.

CONVERSIÓN DE TIPOS DE PUNTEROS (*casting*)

Una manera de acceder al valor apuntado por un puntero genérico es haciendo una conversión de tipos: cambiar el puntero genérico por un puntero al tipo del dato que deseamos decodificar. El siguiente código logrará imprimir el valor de la variable `var` a partir de `ptr`:

```
int var = 1984;
void *ptr = &var;
printf("%d", *((int*)ptr)); // imprime 1984
```

Antes de desreferenciar `ptr` primero lo transformamos al tipo `int*`, haciendo `(int*)ptr`. Esta nueva dirección sí puede ser desreferenciada pues ya no es de tipo `void*`. Cuando se desreferencia esta dirección, con `*((int*)ptr)`, el sistema ahora sí sabe que para imprimir el valor solicitado tiene que decodificar 4 bytes. Hablamos de 4 bytes porque los datos de tipo `int` ocupan ese espacio en memoria. Con la conversión `(int*)` le hemos hecho saber al sistema que lo que se encuentra en la dirección `ptr` es un dato entero.

ARREGLO DE PUNTEROS

Es posible declarar arreglos que almacenen varios punteros del mismo tipo haciendo:

```
int a, b, c;
int* ptrs[3] = {&a, &b, &c};
```

El arreglo `ptrs` contiene las direcciones de memoria de las variables `a`, `b` y `c`. El tamaño que ocupará en memoria, `sizeof(ptrs)`, será 24 bytes (el tamaño de un puntero, 8 bytes, multiplicado por el número de elementos del arreglo, 3).

Podemos referirnos a las variables `a`, `b` y `c` usando expresiones como `*ptrs[0]`, `*ptrs[1]` y `*ptrs[2]`, respectivamente.

Si la inicialización de un arreglo de punteros no incluyera todos los elementos, los elementos no inicializados serían asignados automáticamente con el valor nulo, `NULL`. O sea, la asignación:

```
int* ptrs[3] = {&a, &b};
```

dejaría al tercer elemento, `ptrs[2]`, con el valor `NULL`, una retahíla de 64 bits nulos.

También es posible almacenar punteros de diferentes tipos en un arreglo de punteros genéricos, `void* []`, escribiendo el siguiente código:

```
int a; float b; char c;
void* ptrs[3] = {&a, &b, &c};
```

La diferencia es que en este último caso se necesitaría hacer una conversión de tipos (casting) para poder hacer referencia a las variables `a`, `b` y `c` a través de `ptrs`.

El siguiente código muestra cómo asignar los valores 1984, 3.1416 y 'F' a las variables `a`, `b` y `c`, respectivamente, a través del arreglo de punteros genéricos `ptrs`:

```
*((int*) ptrs[0]) = 1984;
*((float*) ptrs[1]) = 3.1416;
*((char*) ptrs[2]) = 'F';
```

También es posible declarar matrices de punteros con la siguiente sintaxis:

```
int a, b, c, d;
int* ptrs[2][2] = {{&a, &b}, {{&c, &d}}};
```

En este caso una expresión como `*ptrs[1][0]=1000` asignaría 1000 a la variable `c`, cuya dirección se encuentra en la segunda fila y primera columna de la matriz `ptrs`. Como puede deducirse de la expresión anterior, los corchetes [] tienen mayor precedencia que el operador de desreferencia, *.

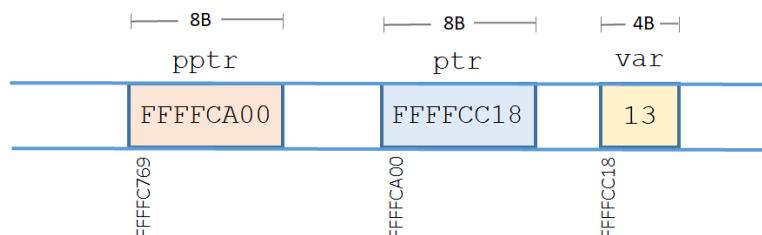
PUNTEROS A PUNTEROS

Así como es posible almacenar la dirección una variable entera o de una variable flotante también es posible guardar la dirección de un puntero. La dirección de un puntero debe ser almacenada en un tipo de datos llamado **puntero a puntero**.

En el siguiente código la variable `pptr` es un puntero a puntero que almacena la dirección del puntero `ptr`. Note que se necesita un doble asterisco, `**`, para declarar punteros a punteros.

```
int var = 13;
int *ptr = &var;
int **pptr = &ptr;
```

Así podría quedar la memoria luego de las declaraciones anteriores:



Los punteros ocupan 8 bytes y las variables enteras ocupan 4 bytes. Todas las direcciones son arbitrarias y sólo se han anotado con fines explicativos.

Primero debemos decir que tanto `ptr` como `pptr` almacenan direcciones de memoria y, por lo tanto, ambos son punteros. La diferencia radica en el tipo de puntero que es cada uno. Como `ptr` almacena la dirección de un entero entonces `ptr` es un puntero a entero, `int*`. Por otro lado, como `pptr` almacena la dirección de `ptr` (la dirección de un puntero a entero), entonces `pptr` sería un puntero a un puntero a entero, `int**`.

Fíjese en el gráfico anterior. Cuando desreferenciamos `pptr` una vez, `*pptr`, obtenemos el dato almacenado en la dirección `pptr` (`FFFFCA00`), que resulta ser la dirección `FFFFCC18`. Para obtener el valor almacenado en `FFFFCC18`, o sea el 13, tendríamos que desreferenciar `*pptr`, lo que se consigue anteponiéndole un segundo asterisco y llegando así a la expresión `**pptr`.

Las tres maneras distintas de imprimir el valor de la variable `var`, 13, serían:

```
printf("%d", var);
printf("%d", *ptr);
printf("%d", **pptr);
```

Las tres maneras distintas de imprimir el dato contenido en `ptr` (la dirección `FFFFCC18`) serían:

```
printf("%p", &var);
printf("%p", ptr);
printf("%p", *pptr);
```

Las dos maneras distintas de mostrar el dato contenido en `pptr` (la dirección FFFFCCA0) serían:

```
printf("%p", &ptr);
printf("%p", pptr);
```

Aunque pueda parecer una dificultad innecesaria, el uso de punteros a punteros aparece de manera natural cuando decidimos gestionar manualmente ciertas regiones de la memoria, tal como veremos en capítulos posteriores.

PUNTEROS CONSTANTES Y PUNTEROS A DATA CONSTANTE

Todos los punteros que hemos estudiado hasta el momento caen dentro de la definición de **punteros variables a data variable**. Este tipo de punteros permite cambiar tanto el valor del puntero como el valor de la variable apuntada. Esta excesiva flexibilidad podría ser indeseable en algunos casos, p.ej. cuando se debe trabajar con datos de sólo lectura y, accidentalmente, por un mal manejo de punteros, uno podría terminar modificando los datos. Por ello existen cuatro clases de punteros; cada una permite restringir lo qué podría o no hacer el programador con un puntero.

	Data variable	Data constante
Puntero variable	<pre>int va = 10; int *pa = &va;</pre>	<pre>int vb = 10; const int *pb = &vb;</pre>
Puntero constante	<pre>int vc = 10; int * const pc = &vc;</pre>	<pre>int vd = 10; const int * const pd = &vd;</pre>

Observe la declaración del puntero `pb` en la celda superior derecha de la tabla anterior. Este puntero es un **puntero variable a data constante**. Uno podría cambiar el puntero, reasignarlo para que apunte a otra dirección, p.ej. `pb=&vx`; pero nunca podría cambiar el valor del dato apuntado por `pb`. Una instrucción como `*pb=20` arrojaría un error en tiempo de compilación.

El puntero `pc` es un **puntero constante a data variable**. Uno podría cambiar el valor del dato apuntado (p.ej. `*pc=20` duplicaría el valor inicial de `vc`); pero nunca podría cambiar el valor del puntero. El puntero siempre apuntará a la dirección con que fue inicializado. Una instrucción como `pc=&vx` después de la inicialización haría que el compilador arroje un error.

El puntero `pd` es un **puntero constante a data constante**. Es el más restrictivo de los cuatro tipos de punteros. No permite alterar el valor de `pd` ni el dato apuntado por este, `*pd`. Ambos intentos conducirían a un error en tiempo de compilación.

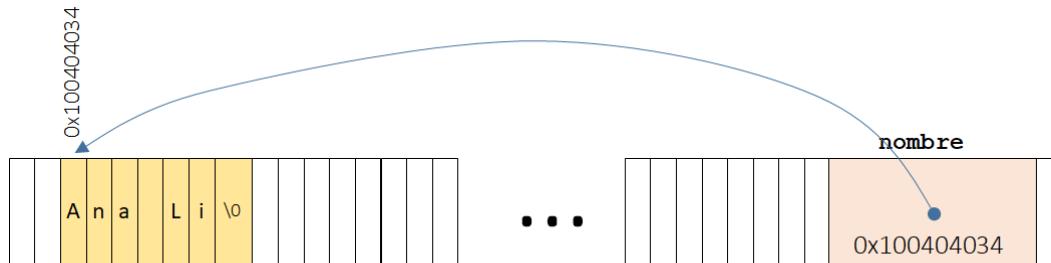
CADERAS Y PUNTEROS A CARÁCTER

Varios estudiantes suelen quedar desconcertados cuando se encuentran por primera vez con una expresión como la siguiente:

```
char *nombre = "Ana Li";
```

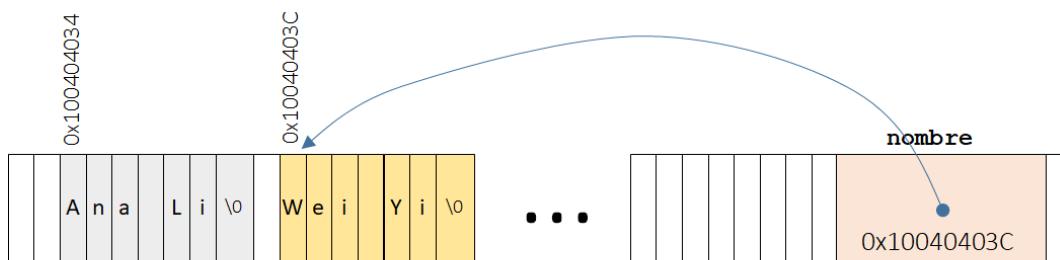
Aparentemente hay una contradicción en esta instrucción. Por un lado, la variable `nombre` ha sido definida como un puntero; por otro lado, `nombre` no está siendo asignada con una dirección de memoria, sino con una cadena, con un texto. Sin embargo, no es incorrecta la expresión anterior. Lo que ocurre es lo siguiente: Al momento de la ejecución el sistema reserva 8 bytes para el puntero `nombre` e inmediatamente le asigna una dirección de memoria, la dirección de la cadena "Ana Li", que ya ha sido previamente escrita en otra zona de la memoria por el mismo sistema, no en los 8 bytes reservados para `nombre`, sino en otro segmento de memoria.

La instrucción anterior dejaría la memoria más o menos de la siguiente forma:



La zona de memoria donde el sistema almacena la cadena "Ana Li" es de sólo lectura. No podemos alterar el contenido de esas celdas, p.ej. cambiar Ana por Ada con `* (nombre+1) = 'd'`. En este caso el sistema lanzaría un error de **violación de segmento** (*segmentation fault*) en tiempo de ejecución.

Sin embargo, no habría problemas si reasignamos el puntero con otra cadena, p.ej. `nombre="Wei Yi"`. En este caso la memoria se actualizaría, más o menos, del siguiente modo:



La cadena apuntada desde `nombre` siempre será colocada en una zona de memoria de sólo lectura.

AGRUPAMIENTO DE CADENAS (*string pooling*)

Todas las cadenas idénticas que utilizamos dentro de un mismo archivo son agrupadas por *GNU gcc*. Por ejemplo, en el programa siguiente, la cadena "Pia" es almacenada una sola vez en memoria, lo cual puede verificarse imprimiendo los valores de s1, s2 y s3, para verificar que todas apuntan a una misma dirección.

```
#include <stdio.h>

char *s3 = "Pia";

void foo(void) {
    char *s2 = "Pia";
    printf("s2=%p\n", s2);
}

int main(void)
{
    char *s1 = "Pia";

    puts("Abajo se muestran tres direcciones iguales");
    printf("s1=%p\n", s1);
    foo();
    printf("s3=%p\n", s3);
}
```

Esta es una opción de optimización que los compiladores implementan y aplican por defecto. Existen parámetros de compilación que pueden (des)activar el agrupamiento de cadenas en ciertos casos.

PROBLEMAS RESUELTOS

1. Manejo de cadenas

Determine lo que ocurre al tratar de copiar, `strcpy(s1, s2)`, concatenar, `strcat(s1, s2)`, o comparar, `strcmp(s1, s2)`, las cadenas `s1` y `s2` definidas como:

```
char *s1 = "Maria", *s2 = "Ana";
```

Diga también qué habría ocurrido si las cadenas se declaraban como arreglos de caracteres

```
char s1[10] = "Maria", s2[10] = "Ana";
```

Los prototipos de las funciones, `strcpy`, `strcat` y `strcmp` están en `string.h`

Solución:

- Primero discutamos la función de copia `strcpy(s1, s2)`. Según su definición, esta recibe como argumentos dos punteros de tipo `char*`, `s1` y `s2`. Su trabajo consiste en copiar los caracteres de la cadena `s2` a partir de la dirección anotada en `s1`.

Cuando `char *s1 = "Maria", *s2 = "Ana"`, el programa lanza un error en tiempo de ejecución (violación de segmento) al intentar ejecutar `strcpy(s1, s2)`. Esto se debe a la imposibilidad de escribir los caracteres '`A`', '`n`', '`a`', '`\0`' en la dirección `s1`. Como vimos en la sección previa, `s1` apunta a una zona de sólo lectura, sobre la que no se puede escribir ningún dato.

Cuando `char s1[10] = "Maria", s2[10] = "Ana"`, el programa funciona correctamente. Luego de la llamada `strcpy(s1, s2)`, la cadena `s1` tomará el valor "`Ana`". En este caso no hay error porque los arreglos `s1` y `s2` son de lectura/escritura. Sólo cuando se asigna una cadena a un puntero a carácter, `char*`, la cadena es almacenada en una zona de sólo lectura.

- Ahora comentemos la función de concatenación `strcat(s1, s2)`. Según su definición esta función recibe como argumentos dos punteros, `s1` y `s2`. Su trabajo es añadir todos los caracteres de `s2` al final de la cadena que se encuentra en la dirección `s1`.

Cuando `char *s1 = "Maria", *s2 = "Ana"`, el programa lanza un error en tiempo de ejecución, justo en la línea `strcat(s1, s2)`. Nuevamente, no se pueden escribir los caracteres '`A`', '`n`', '`a`', '`\0`' inmediatamente después de "`Maria`". No es sólo la dirección `s1` la que está prohibida de escribir, sino toda la región de memoria alrededor de `s1`. Al igual que antes, el error será uno de violación de segmento (*segmentation fault*).

En cambio, si `char s1[10] = "Maria" y s2[10] = "Ana"` el programa se ejecuta con normalidad. Luego de `strcat(s1, s2)`, la cadena `s1` queda como "`MariaAna`". Es importante mencionar que si la cadena concatenada hubiese tenido más de 10 caracteres no se habría recibido ningún error, ni en tiempo de compilación ni en tiempo de ejecución. Sin embargo, aquello sería una muy mala práctica puesto que los caracteres que exceden los 10 bytes reservados para `s1` podrían sobreescribir

otras variables del programa. Este es el conocido problema de desbordamiento de cadena (*buffer overflow*) que discutiremos en capítulos posteriores.

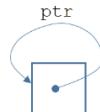
- Finalmente, por definición, la función de comparación de cadenas `strcmp(s1, s2)` retorna cero si las cadenas `s1` y `s2` son iguales, u otro valor en caso contrario. Esta función evalúa que los caracteres de `s1` coincidan con los de `s2`.

Cuando `char *s1 = "Maria", *s2 = "Ana"`, la función se ejecuta sin problemas retornando el valor de 12 (la diferencia entre los códigos ASCII de M y A). La ejecución de `strcmp` no lanza ningún error debido a que esta función en ningún momento intenta actualizar `s1` y `s2`, sólo los lee para poder compararlos.

El mismo resultado se obtiene cuando se utilizan arreglos de caracteres.

Vale mencionar aquí que no es lo mismo comparar el contenido de dos cadenas que comparar sus direcciones. Ya sea que `s1` y `s2` estén declarados como `char*` ó `char[]`, la comparación `s1==s2` es una comparación de direcciones. Sólo retorna verdadero si ambas apuntan a la misma celda. En cambio, la comparación `strcmp(s1, s2)==0` retornará verdadero si el texto grabado en `s1` es el mismo que el de `s2`, aunque estén grabados en direcciones distintas.

2. **El puntero narcisista.** Definir un puntero que se apunte a sí mismo, i.e. que almacene su propia dirección de memoria



Solución:

No sería correcto ceder ante la primera tentación y hacer algo como:

```
int* ptr;  
ptr = &ptr;
```

Las dos líneas anteriores son contradictorias. La primera asegura que `ptr` almacenará la dirección de un entero; la segunda asigna a `ptr` no la dirección de un entero, sino la dirección de un puntero a entero. La manera correcta de resolver este conflicto es haciendo:

```
void* ptr;  
ptr = &ptr;
```

Con un puntero genérico ya no hay contradicción. La primera línea dice que `ptr` almacenará una dirección, no se dice de qué, cualquier dirección. La segunda línea asigna a `ptr`, efectivamente, una dirección. Para verificar que el puntero `ptr` se apunta a sí mismo basta con verificar que las dos direcciones que se imprimen a continuación son iguales:

```
printf("En %p se almacena %p", &ptr, ptr);
```

3. **Todas las celdas de una variable.** Implementar un programa que imprima las direcciones de las ocho celdas reservadas para una variable `var` de tipo `long`.

Solución:

Dado que las celdas ocupadas por una variable siempre son adyacentes bastaría con conocer la dirección del primero de los ocho bytes reservados para la variable `var`. A partir de esta dirección podremos luego movernos por la memoria dando pasos de un byte de longitud.

El código siguiente implementa la funcionalidad requerida:

```
int main(void) {
    long var;
    void* ptr = &var;
    for(int i=0; i<sizeof(long); i++)
        printf("%p\n", ptr+i);
}
```

La variable `var` ocupará 8 bytes en memoria por ser de tipo `long`. La expresión `&var` es la dirección del primero de esos 8 bytes. Esta dirección es almacenada en un puntero genérico. Así aprovechamos la aritmética de los punteros genéricos, de modo que al hacer `ptr+1, ptr+2, ptr+3`, etc. estaríamos caminando por la memoria dando pasos de un byte de longitud.

Dentro del bucle mostrado el programa logra imprimir las ocho direcciones solicitadas usando expresiones como `ptr+i`, `i=0,1,...,7`. Tenga en cuenta que un puntero genérico no puede ser desreferenciado pero sí puede ser impreso. Siempre es posible imprimir un puntero genérico; lo que no puede hacerse (a menos que utilicemos un *casting*) es imprimir el valor al que este apunta.

Note que la variable `var` ni siquiera fue inicializada. Pudo haber contenido cualquier valor inicial. Esto es irrelevante para el problema propuesto, puesto que sólo se nos pide las direcciones de las celdas que ocupa `var`, no el contenido de las mismas.

Finalmente, vale la pena comentar que hubiera sido un error declarar `ptr` como:

```
long *ptr=&var;
```

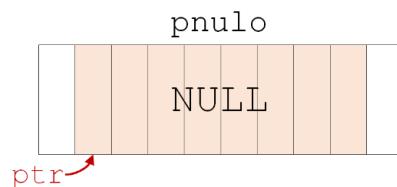
En ese caso habría sido imposible utilizar `ptr` para apuntar a las ocho celdas ocupadas por `var`. Cada salto a partir de `ptr` hubiera sido de 8 bytes. Las expresiones `ptr+1, ptr+2, etc.` apuntarían demasiado lejos, muchos bytes después de `&var`, no a las ocho celdas cuyas direcciones se nos pide mostrar.

4. **Representación interna de NULL.** Mostrar el valor contenido en cada una de las ocho celdas de un puntero inicializado con NULL.

Solución:

El puntero inicializado a NULL es representado como `pnulo` en el gráfico adjunto. Así, lo que se nos pide es el contenido de cada una de las ocho celdas sombreadas.

La estrategia para resolver este problema consiste en definir un segundo puntero, `ptr`, que inicialmente apuntará al primer byte de `pnulo`. Luego el puntero `ptr` será desplazado byte por byte hacia la derecha, ocho veces. Cuando se visite el i -ésimo byte ($i=0,1,\dots,7$), se deberá imprimir el contenido de $*(\text{ptr}+i)$.



En cuanto a los detalles de implementación, el tipo del puntero `pnulo` es indiferente. El valor NULL siempre se representa de la misma forma, ya sea que se almacene en un puntero a entero, a flotante, o a carácter. Lo que debemos definir con cuidado es el tipo de `ptr`.

Existen tres tipos para `ptr` que nos permitirían caminar byte por byte sobre las ocho celdas de `pnulo`. Estos son: puntero genérico (`void*`), puntero a carácter (`char*`), puntero a carácter sin signo (`unsigned char*`). De estas tres opciones mencionadas rápidamente descartamos definir `ptr` como puntero genérico, `void*`. Sabemos que en algún momento necesitaremos imprimir $*(\text{ptr}+i)$, lo cual será imposible si `ptr` fuese genérico, no podríamos desreferenciarlo.

Una posible solución del problema se muestra a continuación:

```
int main(void) {
    int *pnulo = NULL;
    char *ptr = (void*) &pnulo;
    for(int i=0; i<8; i++)
        printf("%02X\n", * (ptr+i));
}
```

`pnulo` fue declarado de tipo `int*`, aunque la salida sería la misma si se usaba `float*`, `char*`, etc..

La expresión `&pnulo` es la dirección de la primera celda de `pnulo`. Esta dirección no puede almacenarse directamente en `ptr`. Una asignación como `char* ptr = &pnulo` causaría que el compilador arroje un error por tipos de punteros incompatibles. Por la forma en que ha sido

declarado `ptr`, este debe recibir la dirección de un carácter (`char*`), no la dirección de un puntero a entero (`int**`), que es el tipo de `&pnulo`. Felizmente esto se soluciona con la conversión de tipos

```
char* ptr = (void*)&pnulo;
```

Gracias a la conversión (*casting*), la dirección `&pnulo` deja de ser del tipo `int**` y pasa a ser considerada como `void*`. Así ya puede ser asignada a cualquier puntero; en particular, a `ptr`.

Finalmente, el contenido de cada celda visitada ha sido impresa en formato hexadecimal, usando dos dígitos hexadecimales, `%02X`, para representar cada byte.

La salida del programa mostrado será `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`. Cada `00` hexadecimal representa los ocho bits nulos que ocupan cada celda de `pnulo`. De esto se verifica que `NULL` es representado internamente como una retahíla de 64 bits (8 bytes) nulos.

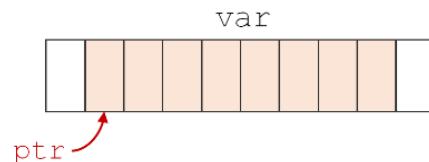
El programa también funciona si definimos `ptr` de tipo `unsigned char*`.

5. Explorador de memoria, memexplorer.c

En el capítulo anterior mostramos que era posible utilizar el depurador `gdb` para observar la representación interna de una variable, i.e. el contenido de cada una sus celdas. También se dijo que dicha tarea era viable implementando un programa que manipule punteros. Se le pide escribir tal programa, uno que imprima el contenido de cada byte ocupado por `var`, una variable que podría ser de cualquier tipo y contener cualquier valor.

Solución:

Haciendo referencia al gráfico que se muestra abajo, la idea general que nos permitirá obtener la representación interna de `var` sería declarar un puntero `ptr` que apunte a la primera celda de `var`. Luego dicho puntero deberá desplazarse por cada una de los celdas ocupados por `var`, usando algo como: `ptr+i`, $i=0, 1, \dots, \text{sizeof}(\text{var})$. Finalmente, cada vez que se visite una celda se deberá desreferenciar el valor de la misma, `* (ptr+i)`, e imprimir dicho valor en hexadecimal, `%X`, tal como lo hace `gdb`.



Para el caso de `float var = -180.625`, su representación interna puede hallarse así:

```

#include <stdio.h>
int main(void) {
    float var = -180.625;
    unsigned char *ptr = (void*) &var;
    for(int i=0; i<sizeof(var); i++)
        printf("%p: %02X\n", ptr+i, *(ptr+i));
}

```

memexplorer.c

En la segunda línea del `main`, la variable `ptr` se ha definido, por conveniencia, como un puntero al tipo `unsigned char`. Dado que este tipo ocupa un byte de memoria, las expresiones `ptr+i` nos permitirán movernos byte por byte sobre la memoria. El puntero `ptr` debe ser inicializado con el primer byte de la zona de memoria que deseamos inspeccionar; es decir, con el primer byte de `var`, que se obtiene con `&var`.

Dado que `&var` es la dirección de un número flotante, `float*`, no puede ser asignada directamente a `ptr`, de tipo `unsigned char*`. Una asignación como `unsigned char* ptr=&var` generaría un problema de incompatibilidad de punteros. Por ello, siempre en la segunda línea del `main`, se hace una conversión de tipos. La dirección `(void*) &var` ya es genérica y, como tal, sí puede ser asignada a un puntero de cualquier tipo; en particular, a uno de tipo `unsigned char*`.

La zona de memoria que deseamos inspeccionar contiene tantas celdas como bytes ocupa la variable `var`. Las celdas de interés son visitadas una por una dentro del bucle:

```
for(int i=0; i<sizeof(var); i++)
```

En el momento que se visita la i -ésima celda ocupada por `var` se imprimen tanto su dirección como el valor contenido en esta; es decir, `ptr+i` y `*ptr+i`, respectivamente. El formato `%02X` usado para imprimir el contenido de cada celda hace que dicho contenido sea impreso con dos dígitos hexadecimales, rellenando con ceros a la izquierda cuando sea necesario.

La salida del programa anterior en una arquitectura *little-endian* es la siguiente:

```
User@DESKTOP-ETMGBI ~
$ ./a
0xfffffcc0c: 00
0xfffffcc0d: A0
0xfffffcc0e: 34
0xfffffcc0f: C3
```

El programa seguirá funcionando correctamente aunque se cambie el valor asignado a `var` o, incluso, el tipo de la variable `var`.

No hubiera sido lo mismo si se definía `ptr` como `char*`, sin el modificador `unsigned`. En dicho caso la salida hubiera sido errónea, tal como se muestra abajo.

```
User@DESKTOP-ETMGBI ~
$ ./a
0xfffffcc0c: 00
0xfffffcc0d: FFFFFFA0
0xfffffcc0e: 34
0xfffffcc0f: FFFFFFFC3
```

La diferencia radica en la interpretación del primer bit de cada celda. Si `ptr` se define como `char*`, el primer bit de cada celda sería interpretado como bit de signo. En ese caso, el contenido de algunas celdas, p.ej. `A0=1010 0000`, sería interpretado erróneamente como un entero negativo a causa de que su primer bit es 1. Las operaciones de transformación que se realizan para decodificar dicho número negativo generarían errores similares al mostrado en el último gráfico. En cambio, cuando usamos `unsigned char*` dejamos sobreentendido que el valor apuntado por `ptr` no tiene signo. De ese modo el programa imprime los 8 bits de cada celda directamente, de binario a hexadecimal. No interpreta el primer bit de cada celda, ni realiza operaciones de complemento a dos en su intento por decodificar su valor.

- Determine manualmente cuál será la salida del siguiente programa cuando se ejecute sobre una arquitectura *little-endian*.

```
int main(void) {
    unsigned char c = 150;
    printf("%d", *((char*)&c));
}
```

Solución:

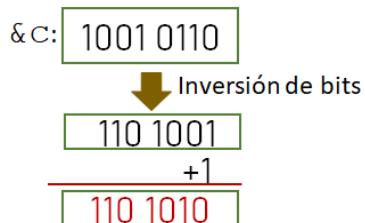
Se debe empezar aclarando que el programa mostrado carece de utilidad práctica. Es intencionadamente artificioso y sirve como ejercicio para entrenar a los neófitos en el espinoso camino de los punteros.

En la primera línea del programa, el sistema reservará 8 bits para almacenar la variable `c`. Internamente, estos 8 bits coinciden con la representación binaria de 150; es decir:

```
1001 0110
```

La segunda línea imprime un número decimal. Dicho número es la cadena de bits almacenada en `&c`, pero interpretada como si fuera de tipo `char`, no `unsigned char`. En otras palabras, a pesar de que la variable `c` fue declarada como `unsigned char`, luego será leída como si fuera de tipo `char`.

Cuando los 8 bits mostrados anteriormente son interpretados como `char` el primer bit juega un papel importante. En nuestro caso, este bit es 1. Esto significa que el supuesto dato `char` que estaría representado en esos 8 bits sería un valor negativo. Para desvelar la magnitud de dicho valor se debe obtener el complemento a dos, $C-2$, de los 7 bits restantes. Esto se hace del siguiente modo:



Tal como muestran los cálculos, el valor negativo buscado sería $-110\ 1010$ que, en base decimal, %d, es -106. Esa será justamente la salida del programa.

Como se ve en este ejercicio, la cadena de bits que representa al número 150 de tipo `unsigned char` es la misma que se usa para representar a -106 cuando se almacena como `char`.

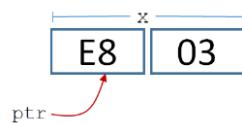
El resultado habría sido el mismo si la computadora hubiera tenido arquitectura *big-endian*.

- Determine manualmente cuál será la salida del siguiente programa cuando se ejecute sobre una arquitectura *little-endian*.

```
int main(void) {
    short x = 1000;
    char *ptr = (void*) &x;
    printf("%d", *ptr);
}
```

Solución:

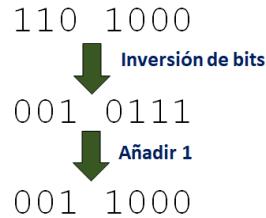
Empecemos determinando cómo se almacena x en la memoria. Usando una calculadora se puede saber que el entero 1000 en hexadecimal se escribe como 3E8. Además sabemos que las variables de tipo `short` ocupan 2 bytes. Dado que se está asumiendo una arquitectura *little-endian*, la representación interna de la variable x sería:



El gráfico ya muestra que el puntero `ptr` apuntará a la primera celda de `x`. Esto debido a la instrucción: `char* ptr = (void*) &x;` Esta última asignación es correcta y, gracias a la conversión `(void*)`, el compilador no arroja ningún error de incompatibilidad de punteros.

En la última línea el programa imprime `*ptr`, el valor codificado como E8. Sólo se decodificará un byte porque `ptr` ha sido definido como `char*`. En cuanto a la decodificación de E8 = 1110 1000, se observa que éste tiene a la unidad como bit de signo. Esto significa que para el sistema el número apuntado por `ptr` es negativo. Sólo nos falta saber cuál es la magnitud de este número negativo. Para esto se debe aplicar el complemento a 2 a la cadena 110 1000 (sin el bit de signo). Como ya se

explicó antes, este proceso consta de dos pasos: inversión de bits y adición de una unidad, tal como se muestra abajo.



El valor binario obtenido, 001 1000, es la magnitud del entero negativo que estamos intentando desentrañar. Éste, al ser impreso en formato decimal, `%d`, aparecerá como `-24`, que es la salida del programa mostrado.

Vale la pena ser explícito en mencionar que así como el complemento a dos convierte la representación de un entero positivo en la representación de su contraparte negativa, lo opuesto también es cierto. Para conocer qué negativo se esconde tras una cadena de bits se le debe aplicar el complemento a dos. Así se interpreta aquella propiedad $C-2(C-2(x)) = x$, que ya se mostró anteriormente, pero que por primera vez se está comentando.

- Determine cuál será la salida del siguiente programa sobre una arquitectura *little-endian*.

```

int main(void) {
    int var = 0xA5BFF7C;
    short *ptr = (void*) &var;
    printf("%d", *ptr);
}
  
```

Solución:

La primera línea reservará cuatro bytes de memoria para la variable entera `var`. En dichos bytes se almacenará un dato que, para suerte nuestra, ya está dado en formato hexadecimal. Sin tener que hacer cambios de base notamos que la representación de `var` en *big-endian* sería: 0A 5B FF 7C. Y en *little-endian* sería:

7C FF 5B 0A

La segunda línea del `main` hace que `ptr` apunte al primer byte de `var`, que contiene 7C.

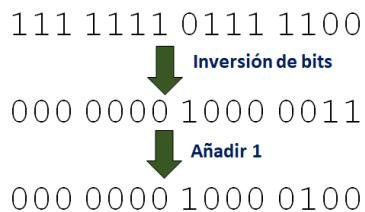
La tercera línea imprimirá el dato apuntado por `ptr` en formato decimal. Note que como `ptr` ha sido definido como puntero a entero corto (`short*`), la expresión `*ptr` hará que el sistema decodifique el número representado como 7C FF, un dato de dos bytes, el tamaño del tipo `short`.

Una arquitectura *little-endian* asumirá que el primer byte del dato 7C FF es FF, y el segundo, 7C. Esto es debido a que las computadoras con arquitectura *little-endian* leen los bytes de derecha a izquierda.

Al mostrar FF 7C en binario se nota que el primer bit, el bit de signo, es 1 y, por lo tanto, el valor que decodificará el sistema será un número negativo.



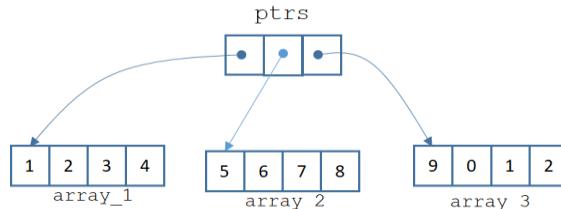
Para conocer la magnitud de este negativo se debe obtener el complemento a dos de la cadena anterior, sin el bit de signo. Esto se hace así:



En formato decimal, %d, -1000 0100₂ es -132, que es la salida del programa.

9. Arreglo de punteros.

Implementar la estructura mostrada en la figura y mostrar los elementos de array_1, array_2 y array_3, sin hacer ninguna referencia directa a estos arreglos. Sólo debe utilizarse el arreglo ptrs, que contiene las direcciones de los primeros elementos de array_1, array_2 y array_3



Solución:

Para implementar la estructura mostrada se debe declarar lo siguiente:

```
int array_1[4] = {1, 2, 3, 4};
int array_2[4] = {5, 6, 7, 8};
int array_3[4] = {9, 0, 1, 2};
```

Recuerde que, por decaimiento, array_1 se refiere a la dirección de su primer elemento, &array_1[0]. Entonces, tanto array_1 como array_2 y array_3 son direcciones de enteros y, por lo tanto, deben ser almacenados en un arreglo de punteros a enteros, ptrs.

```
int *ptrs[3] = {array_1, array_2, array_3};
```

Usando desreferenciación y aritmética de punteros es posible hacer:

```
*array_1, *(array_1 + 1), *(array_1 + 2) y *(array_1 + 3)
```

para acceder a los cuatro elementos de `array_1`. Y análogamente para `array_2` y `array_3`.

Pero también puede usarse `ptrs[0]` en lugar de `array_1`; `ptrs[1]` en lugar de `array_2` y `ptrs[2]` en lugar de `array_3`.

La solución al problema propuesto sería así:

```
int main(void) {
    int array_1[4] = {1,2,3,4};
    int array_2[4] = {5,6,7,8};
    int array_3[4] = {9,0,1,2};

    int *ptrs[3] = {array_1, array_2, array_3};
    for(int i=0; i<3; i++)
        for(int j=0; j<4; j++)
            printf("%d ", *(ptrs[i]+j));
}
```

El bucle exterior permite visitar, uno por uno, cada elemento de `ptrs`. Cuando se visita el elemento `ptrs[i]` ($i=0,1,2$), se aprovecha para inmediatamente imprimir todos los elementos que se encuentran adyacentes a dicha dirección; es decir, en las direcciones `ptrs[i]+j`, $j=0,1,2,3$. Estos valores son justamente los datos de `array_1`, `array_2` y `array_3`.

PROBLEMAS PROPUESTOS

1. Determinar manualmente cuál sería la salida de cada uno de los siguientes fragmentos de código.
Asuma que se ejecutan en una computadora con arquitectura *little-endian*:

```
int x = 0xAABBCCDD;
char *p = (void*) &x;
printf("%d", *(p+2));
```

```
char arr[] = {1,2,3,4};
short *ptr = (void*) &arr[2];
printf("%d", *ptr);
```

```
int M[4][2] = {{1}};
const int * const ptr = &M[0][0];
for(int i=0; i< sizeof(M[0]); i++)
    printf("%d ", *(ptr+i));
```

```
short x = -280;
char *p = (void*)&x;
printf("%d", *(p+1));
```

2. Usando únicamente el arreglo ps, complete el código para que imprima lo que se pide:

```
void main(void) {
    char *str = "Juan";
    int x = 5;
    short arr[4] = {10, 20, 30, 40};

    void* ps[4] = {str, &x, arr, &duplica};

    printf("%s", [REDACTED]); // prints Juan
    printf("%d", [REDACTED]); // prints value of x
    printf("%d", [REDACTED]); // prints 3rd element of arr
}
```

3. Complete el siguiente código de modo que los datos de names1 se transfieran a names2. Luego imprimir los nombres mostrados desde names2

```
int main(void) {
    char *names1[3] = {"Ana", "Liv", "Katty"};
    char names2[3][10];
    /* Escribir su código debajo de esta línea */
}
```

4. Escriba un programa que imprima el *endiannes* del computador. Para ello podría almacenar el número 1 en una variable de tipo `short` y verificar si su único bit significativo se almacena en el primer o el segundo byte.
5. Escriba un programa que imprima los 256 datos posibles que podrían ser codificados en un byte de memoria. Indique el valor de cada dato cuando este se interpreta como `unsigned char` y como `char`. A continuación se muestran unas pocas líneas de la tabla de equivalencias pedida. La primera columna representa una secuencia de 8 bits; la segunda, el dato de tipo `unsigned char` codificado en esos 8 bits; y la tercera, el dato `char` detrás de la misma secuencia.

7C:	124	+124
7D:	125	+125
7E:	126	+126
7F:	127	+127
80:	128	-128
81:	129	-127
82:	130	-126
83:	131	-125

CAPÍTULO

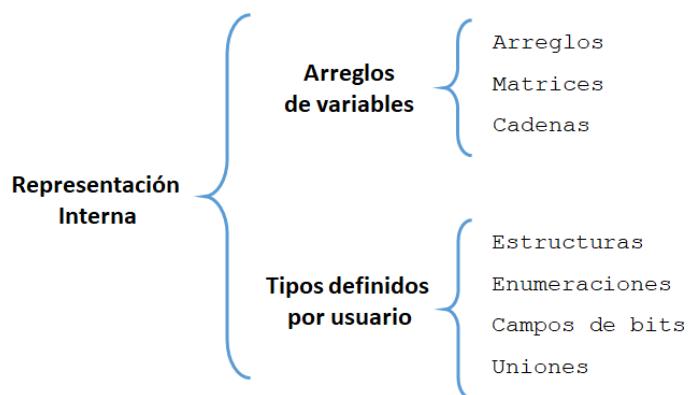
5

REPRESENTACIÓN INTERNA DE DATOS AGREGADOS

Anteriormente vimos cómo estaban codificados los datos atómicos (`short`, `int`, `float`, etc.) en la memoria del computador. Aprendimos a calcular manualmente las representaciones internas de distintos datos según su tipo y según la arquitectura utilizada (*little/big-endian*).

En este capítulo estudiaremos la representación interna de los datos agregados (arreglos, matrices, estructuras, etc.). Dado que un dato agregado es una colección de datos atómicos era indispensable conocer primero la representación interna de estos últimos.

Tal como se muestra abajo, separaremos nuestro estudio en dos secciones:



Primero veremos como se almacenan las colecciones de variables de un mismo tipo, i.e. arreglos, matrices y cadenas. Todos estos tienen una representación que puede calcularse manualmente siempre que se conozca el *endianness* del computador. Posteriormente se estudiará la representación interna de los tipos definidos por usuario. Aunque siempre se puede determinar de manera inequívoca la representación de enumeraciones y uniones, no necesariamente ocurrirá lo mismo con las estructuras y los campos de bits.

REPRESENTACIÓN INTERNA DE ARREGLOS DE VARIABLES

Un arreglo es una colección de variables del mismo tipo agrupadas bajo un único nombre. Los datos de un arreglo se almacenan de forma contigua en la memoria del computador, uno a continuación de otro, sin dejar huecos. Consecuentemente, la representación interna de un arreglo es igual a la secuencia de representaciones internas de cada uno de sus datos.

Dado que las matrices y cadenas son tipos especiales de arreglos, lo anterior también es válido para estos tipos de estructuras.

ARREGLOS

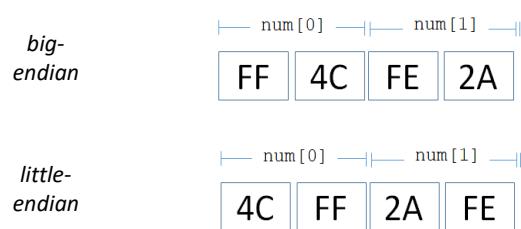
La representación interna de un arreglo se construye a partir de las representaciones internas de sus elementos. Debe considerarse que dentro de la memoria los elementos de un arreglo siempre se almacenan en el orden indicado por sus subíndices.

Ejemplo. Determine la representación interna del arreglo `short num[2] = {-180, -470}`.

Solución:

El arreglo `num` ocupará 4 bytes memoria: en los dos primeros se guardará el entero `-180`; y en los dos últimos, `-470`. Felizmente, del capítulo anterior, ya conocemos las representaciones internas de cada uno de estos datos.

Dado que la representación de un dato depende del *endianness* del computador, la representación interna de un arreglo también dependerá de esta característica. Siendo así, el arreglo `num` podría representarse de cualquiera de las dos siguientes formas:



Es importante que seamos explícitos en mencionar que cuando trabajamos con arreglos la representación en formato *little-endian* no se obtiene invirtiendo los bytes de la representación en *big-endian*, tal como se podía hacer con los datos atómicos. De haber sido así, la representación en formato *little-endian* del arreglo `num` habría sido: `2A FE 4C FF`, lo cual no es correcto, tal como puede comprobarse usando `gdb`. Lo que debe invertirse (siempre a nivel de bytes, no de bits) es la representación de cada elemento del arreglo, no la representación de todo el arreglo en su conjunto.

MATRICES

En términos abstractos una matriz puede ser concebida como un arreglo bidimensional de datos; pero, físicamente, estos datos se almacenan linealmente dentro de la memoria: Primero, los datos de la primera fila; a continuación los de la segunda fila; y así sucesivamente.

Ejemplo. Halle la representación de: `short M[2][3] = {{-180, 10, 0}, {50, 0, -470}}`.

Solución:

Dentro de la memoria los seis elementos de la matriz `M` serán almacenados por filas; es decir, en el siguiente orden:

`M[0][0], M[0][1], M[0][2], M[1][0], M[1][1], y M[1][2]`

Por ser de tipo `short`, cada elemento ocupará dos bytes en memoria. Como ya se conocen las representaciones de `-180` y `-470`, sólo nos faltaría encontrar las de `0`, `10` y `50`.

Sin necesidad de mostrar cálculos manuales, directamente mencionaré las representaciones internas en *big-endian* de cada uno de estos números. El cero se representa como `00 00`; el diez como `00 0A`; y el cincuenta como `00 32`. Se dejan las verificaciones al lector.

La representación de la matriz `M`, dependiendo de la arquitectura, sería una de las siguientes:



CADENAS

Las cadenas son secuencias de caracteres que terminan con el carácter especial '`\0`'. Cada uno de estos caracteres, incluso el carácter de terminación, ocupa un byte de memoria. La representación interna de una cadena es independiente de la arquitectura con que se trabaje.

Ejemplo. ¿Cuál será la representación interna de la cadena `char v[] = "abaco"`?

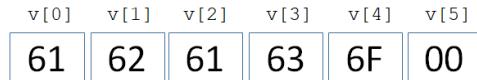
Solución:

La cadena `v` ocupará seis bytes en memoria. En los primeros cinco se almacenarán los caracteres de abaco o, más precisamente, sus códigos ASCII. En el sexto byte se grabará el carácter de terminación, '\0'.

Con una tabla de códigos ASCII se puede verificar que el código del carácter 'a' es 97; el de 'b' es 98; el de 'c', 99; y así sucesivamente. Estos códigos numéricos son los que se almacenan en memoria.

Carácter	Cód. ASCII (decimal)	Cód. ASCII (binario)	Cód. ASCII (hexadecimal)
'a'	97	0110 0001	61
'b'	98	0110 0010	62
'c'	99	0110 0011	63
'o'	111	0110 1111	6F

La representación de la cadena `v` es la siguiente:



Cada dato de la cadena se representa con un único byte y, por lo tanto, da lo mismo si dicho byte se escribe de izquierda a derecha o de derecha a izquierda a nivel de bytes. Consecuentemente, la representación anterior será la misma en cualquier computadora, con cualquier *endianness*.

El último byte mostrado es una secuencia de 8 bits nulos, 0000 0000, porque esa es la representación del carácter de terminación, '\0', tal como se mencionó anteriormente, cuando discutimos la representación interna de valores especiales.

El programa `memexplorer.c`, discutido en la sección de ejercicios del capítulo anterior, puede ser ligeramente modificado para que pueda imprimir la representación interna de arreglos, matrices y cadenas, tal como se muestra abajo:

```
#include <stdio.h>
int main(void) {
    short var[2][3]={{-180, 10, 0},{50, 0, -470}};
    unsigned char *ptr = (void*) &var;
    for(int i=0; i<sizeof(var); i++)
        printf("%p: %02X\n", ptr+i, *(ptr+i));
}
```

El único cambio realizado con respecto al código original consistió en reemplazar la variable flotante `float var=-180.625` por la matriz `short var={{-180, 10, 0}, {50, 0, -470}}.`

REPRESENTACIÓN INTERNA DE TIPOS DEFINIDOS POR USUARIO

Además de los tipos predefinidos que disponemos al instalar las librerías estándar de C el programador también puede definir sus propios tipos de datos. Todo tipo definido por usuario necesariamente pertenece a alguna de las siguientes categorías: estructuras, enumeraciones, campos de bits o uniones.

ESTRUCTURAS

Una estructura es una colección arbitraria de variables bajo un mismo nombre.

Sólo cuando los atributos de una estructura se almacenan en posiciones contiguas de memoria es posible calcular manualmente su representación interna. En tales casos, la representación de una estructura es la secuencia de las representaciones de sus atributos (en el orden en que fueron declarados).

Pero, en general, cuando una estructura es cargada a la memoria existen «huecos» entre las celdas de un atributo y el siguiente. Estos huecos son añadidos automáticamente por el sistema –y, por lo tanto, dependientes del sistema– por cuestiones de optimización, p.ej. para asegurarse que el tamaño de cada estructura sea un múltiplo de 4 bytes. Por esta razón no siempre se puede calcular manualmente la representación interna de una estructura; lo que sí es posible es obtener su representación interna computacionalmente, con un programa como el que se muestra abajo:

```
#include <stdio.h>
struct persona {
    int codigo;
    char sexo;
    int edad;
};
int main(void) {
    struct persona var = {102030, 'M', 46};
    unsigned char *ptr = (void*) &var;
    for(int i=0; i<sizeof(var); i++)
        printf("%p: %02X\n", ptr+i, *(ptr+i));
}
```

La salida del programa se muestra en la página siguiente.

Esta salida revela que la estructura `var` ocupará 12 bytes; es decir, tres bytes más grande que todos sus atributos juntos: `codigo` (4 bytes), `sexo` (1 byte) y `edad` (4 bytes). Estos tres bytes son huecos que el sistema deja dentro del espacio reservado para la estructura. En general, el contenido de estos huecos es impredecible, podrían contener cualquier valor y, por ello, se hace imposible calcular manualmente la representación interna de una estructura.

```
User@DESKTOP-ETMGMBI ~
$ ./a
0xfffffc04 : 8E
0xfffffc05 : 8E
0xfffffc06 : 01
0xfffffc07 : 00
0xfffffc08 : 4D
0xfffffc09 : 00
0xfffffc0a : 00
0xfffffc0b : 00
0xfffffc0c : 2E
0xfffffc0d : 00
0xfffffc0e : 00
0xfffffc0f : 00
```

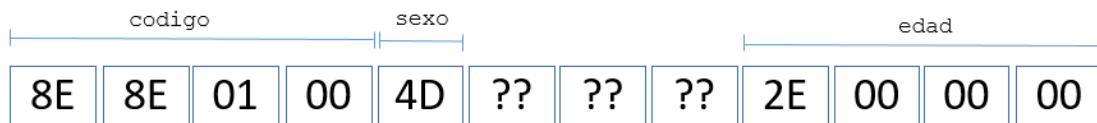
Para poder identificar cuáles de los 12 bytes mostrados en la salida son huecos y, por lo tanto, no contienen información relevante, se puede hacer:

```
printf("%d", offsetof(struct persona, codigo));
printf("%d", offsetof(struct persona, sexo));
printf("%d", offsetof(struct persona, edad));
```

El código anterior imprime 0, 4 y 8, indicando que los atributos `codigo`, `sexo` y `edad` se encuentran almacenados a partir del primer, quinto y noveno byte de la estructura `var`. Como el tamaño de cada atributo es conocido, se deduce que el 6^{to}, 7^{mo} y 8^{vo} byte de la estructura serán huecos, bytes que contienen cualquier valor.

La función `offsetof` tiene su prototipo en `stddef.h`.

La representación de `var` (en *little-endian*), con todos sus huecos, se muestra a continuación:



CAMPOS DE BITS

Los campos de bits son atributos especiales de una estructura. Sus datos pueden ocupar un pequeño número de bits en memoria, según lo que defina arbitrariamente el programador. Varios campos de bits podrían compartir un mismo byte en memoria.

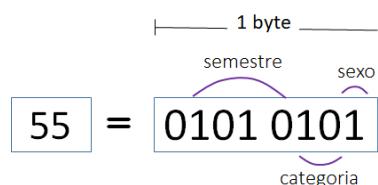
Dado que C no permite acceder a la dirección de un campo de bits es difícil escribir un programa que permita conocer con exactitud cómo y dónde están almacenados.

En la salida del siguiente programa los primeros cuatro bytes (8E 8E 01 00) representan el código del estudiante, o sea el entero 10230. Pero es difícil determinar cómo y dónde se almacenan los campos de bits sexo, categoria y semestre, de 1, 2 y 4 bits, respectivamente.

```
#include <stdio.h>
struct estudiante {
    int codigo;
    unsigned int sexo:1;
    unsigned int categoria:2;
    unsigned int semestre:4;
};
int main(void) {
    struct estudiante var = {102030, 1, 2, 10};
    unsigned char *ptr = (void*) &var;
    for(int i=0; i<sizeof(var); i++)
        printf("%p: %02X\n", ptr+i, *(ptr+i));
}
```

```
User@DESKTOP-ETMGMIBI ~
$ ./a
0xfffffc28: 8E
0xfffffc29: 8E
0xfffffc2a: 01
0xfffffc2b: 00
0xfffffc2c: 55
0xfffffc2d: 00
0xfffffc2e: 00
0xfffffc2f: 00
```

Luego de cierto esfuerzo puede inferirse que el quinto byte, 55, está guardando simultáneamente los tres campos de bits, sexo, categoria y semestre, del siguiente modo:



Los valores 1, 2 y 10 con que se inicializaron los campos `codigo`, `sexo` y `semestre`, respectivamente, están representados en ese quinto byte, como 1, 10 y 1010. Estas cadenas de bits aparecen escritas de izquierda a derecha y concatenadas de derecha a izquierda, de forma contigua; pero no siempre ocurrirá así. En general, la representación interna de los campos de bits no se puede calcular manualmente pues depende de cada compilador.

Note que el espacio reservado para la estructura `estudiante` nuevamente posee huecos: los últimos tres bytes (que van desde `0xfffffc2d` hasta `0xfffffc2f`) no almacenan ningún dato de usuario; son añadidos por el sistema por cuestiones de optimización.

ENUMERACIONES

La representación interna de una variable de tipo enumerado se puede obtener siguiendo las reglas estudiadas para codificar números enteros con signo.

Ejemplo. Dada la enumeración `enum tipo_sangre {A, B, AB, O}`, ¿cómo se almacena internamente la variable `enum tipo_sangre var = AB`?

Solución:

La variable `var` es tratada como si fuera un entero. Ocupará 4 bytes de memoria y se representará como el entero 2, o sea, como 00 00 00 02 en una *big-endian*, o 02 00 00 00 en una *little-endian*. Recuerde que los miembros de una enumeración son constantes enteras cuyos valores consecutivos inician en cero. En este caso, A=0, B=1, AB=2 y O=3.

Puede verificar lo dicho usando el programa `memexplorer.c` para la variable enumerada `var`.

UNIONES

Una unión es una colección de atributos que comparten el mismo espacio de memoria. El tamaño de una unión coincide con el de su atributo más extenso, el que ocupa más bytes en memoria.

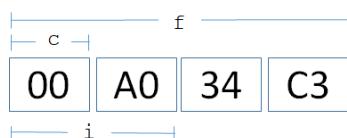
Ejemplo. Halle la representación interna en *little-endian* de la unión `var` mostrada abajo:

```
union Dato {
    float f;
    char c;
    short i;
};

int main(void) {
    union Dato var;
    var.f = -180.625;
}
```

Solución:

La variable `var` es de tipo `union Dato` y tiene tres atributos de tipos `float` (4 bytes), `char` (1 byte) y `short` (2 bytes). La unión `var` ocupará 4 bytes en memoria, porque ese es el tamaño de su atributo más extenso, el flotante `f`. En esos 4 bytes estarán representados simultáneamente los tres atributos de la unión: `f`, `c` y `i`, sobreponiéndose, uno encima de otro, tal como se aprecia en el siguiente gráfico:



Los bytes 00 A0 34 C3 se cargan en memoria a causa de la asignación `var.f=-180.625`. Estos constituyen la representación interna (en *little-endian*) de -180.625, tal como se vio anteriormente.

Todos los atributos de la unión se almacenan a partir del primero de los cuatro bytes mostrados; es decir, las direcciones `&var.f`, `&var.c` y `&var.i` tienen el mismo valor. En el programa mostrado el atributo `c` contiene 00 y el atributo `i` almacena 00 A0.

Si uno alterase el valor de `c`, digamos con `var.c = 'A'`, la nueva representación de la unión sería:

41	A0	34	C3
----	----	----	----

Tome en cuenta que el código ASCII de 'A' es 65 (41 en hexadecimal).

Tal como se observa, el cambio de un atributo altera el valor de los restantes.

PROBLEMAS RESUELTOS

1. **Otro explorador de memoria.** Las uniones pueden ser ingeniosamente utilizadas para obtener la representación interna de un dato sin necesidad de utilizar punteros. Obtenga la representación interna del flotante -180.625 usando uniones. Asuma una arquitectura *little-endian*.

Solución:

Una ingeniosa solución consiste en sobreponer cuatro caracteres en el mismo espacio de memoria donde se almacenará -180.625 , y luego imprimir dichos caracteres. Advertimos que sería una mala idea ceder ante la primera tentación y tratar de implementar la estrategia descrita con algo como:

```
union Flotante {
    float f;
    char c1;
    char c2;
    char c3;
    char c4;
};
```

En este caso, los cuatro caracteres ($c1, c2, c3, c4$) tendrían siempre el mismo contenido, el dato almacenado en el primer byte del atributo f . La implementación correcta sería:

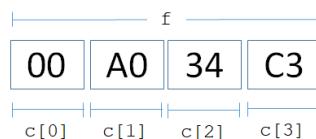
```
#include <stdio.h>

union Flotante {
    float f;
    unsigned char c[4];
};

int main(void) {
    union Flotante var;
    var.f = -180.625;
    for(int i=0; i<sizeof(var); i++)
        printf("%p: %02X\n", &var.c[i], var.c[i]);
}
```

```
User@DESKTOP-ETMGBI ~
$ ./a
0xfffffc38: 00
0xfffffc39: A0
0xfffffc3a: 34
0xfffffc3b: C3
```

Por la definición de `union Flotante` se deduce que `var` ocupará 4 bytes. Estos bytes serán ocupados por `00 A0 34 C3`, la representación interna de -180.625 en *little-endian*. Pero, al ser `var` una unión, esos mismos cuatro bytes serán ocupados también por el arreglo de caracteres `c`, tal como se muestra abajo:



Para mostrar cada byte de `f` el programa sólo tiene que imprimir `var.c[i]`, $i=0, \dots, 3$. No hay necesidad de usar punteros.

Vale comentar que habría sido un error utilizar el tipo `char` (sin el modificador `unsigned`) para definir el arreglo de caracteres `c`. Esto habría causado que el sistema considere el bit de signo de cada `var.c[i]`, lo cual es innecesario. Sólo necesitamos que se imprima lo que está en la memoria en ese momento, el contenido de cada celda. Dicho contenido no es positivo ni negativo y, por lo tanto, se debe evitar que el sistema interprete el signo de cada celda.

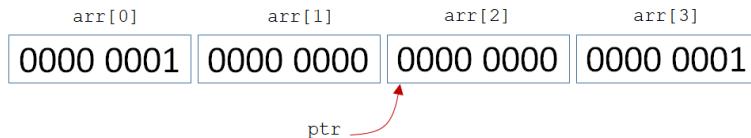
2. **Un arreglo de booleanos.** Determine manualmente cuál será la salida del siguiente programa cuando se ejecute sobre una arquitectura *little-endian*.

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

int main(void) {
    bool arr[4] = {true, false, false, true};
    short *ptr = (void*)&arr[2];
    printf("%d", *ptr);
}
```

Solución:

Primero debemos hallar la representación interna del arreglo `arr`. Este contiene 4 valores booleanos y, por ende, ocupa 4 bytes en memoria. Como ya se mencionó antes, las representaciones internas de `true` y `false` son `00000001` y `00000000`, respectivamente. Siendo así, la representación del arreglo `arr` será la siguiente:



La segunda línea del `main` indica que el puntero `ptr` va a apuntar al tercer elemento del arreglo, `arr[2]`. Aunque `&arr[2]` es de tipo `bool*` y `ptr` fue definido como `short*`, el conversor de tipos (`void*`) conjura el problema de incompatibilidad de punteros.

La tercera línea del `main` imprimirá el valor apuntado por `ptr`. Pero no vaya a creer que este valor es el contenido de la tercera celda. Por la forma en que ha sido definido el puntero `ptr`, el sistema asumirá que `ptr` apunta a un dato de tipo `short` y, por lo tanto, decodificará los dos bytes que se encuentran a partir de `ptr`. Asumirá que las celdas etiquetadas como `arr[2]` y `arr[3]` contienen ambas el dato de tipo `short` que se tiene que imprimir.

Recién aquí debemos considerar el *endianness* de la máquina. Si el programa se ejecutase sobre una *big-endian*, el entero apuntado por `ptr` sería 1, pues `00000000 00000001` es la representación de la unidad en una arquitectura *big-endian*. Pero, como estamos asumiendo una arquitectura *little-endian*, el sistema considerará que los dos últimos bytes no son el entero buscado en binario, sino el resultado de invertir la representación binaria del número buscado a nivel de bytes.

Un sistema *little-endian* decodificará la secuencia `00000001 00000000`. El resultado se imprimirá en formato decimal (%d). La salida del programa será 256.

3. **Desbordamiento de buffer (buffer overflow).** Usando sus conocimientos de representación interna explique el extraño comportamiento del siguiente programa, que es ejecutado sobre una arquitectura *little-endian*. No hay huecos entre los atributos de la estructura.

```
#include<stdio.h>
int main(void) {
    struct {
        char nombre[8];
        int edad;
    } persona;

    printf("Ingrese su edad:");
    scanf("%s", &persona.edad);

    printf("Ingrese su nombre:");
    scanf("%s", persona.nombre);

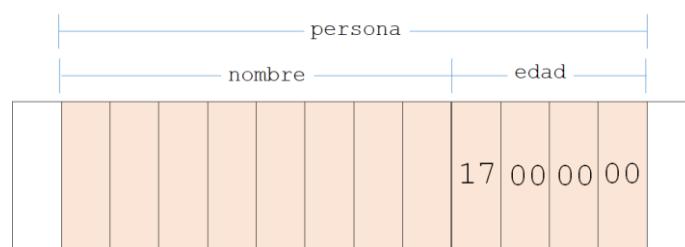
    printf("%s, su edad es %d",
           persona.nombre,
           persona.edad);
}
```

```
User@DESKTOP-ETMGBI ~
$ ./a
Ingrese su edad:23
Ingrese su nombre:Alejandro
Alejandro, su edad es 111
User@DESKTOP-ETMGBI ~
$ ./a
Ingrese su edad:19
Ingrese su nombre:Alexandra
Alexandra, su edad es 97
```

Solución:

Dentro de la función principal se ha declarado una **estructura anónima** y una variable, `persona`, de dicho tipo. Dado que la estructura no tiene nombre ya no podrá crearse ninguna otra variable de ese tipo en el resto del programa. Aclarado esto, debemos continuar nuestro análisis, pues la estructura anónima no es la causa del extraño comportamiento de este programa que, aparentemente, hace envejecer a sus usuarios.

Al no haber huecos en la estructura la variable `persona` ocupará 12 bytes en memoria: los primeros 8 son para alojar el arreglo `nombre`; los 4 siguientes, para almacenar la `edad`. Se puede deducir que luego de que el primer usuario ingresó su edad, 23, las celdas de memoria ocupadas por la variable `persona` quedaron de la siguiente manera:

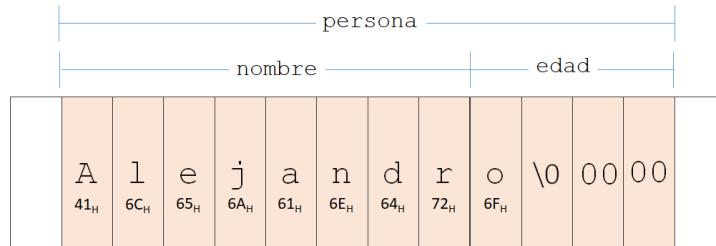


El entero 23 se representa como `17 00 00 00` en *little-endian*.

Hasta ese momento todo marchaba bien. El problema apareció cuando el usuario ingresó luego su nombre que, desgraciadamente, necesitaba más de los 8 bytes que algún programador mezquino creyó suficientes para almacenar dicho dato. El arreglo `nombre` será rebasado; los bytes que no quepan dentro de este arreglo se escribirán en las celdas subsiguientes, sin importar si estas ya contienen datos, como de hecho ocurre en este caso.

La función `scanf` no hace nada para prevenir ni para controlar el derramamiento de datos del arreglo `nombre`. El texto ingresado por el usuario podría desbordarse incluso fuera de la zona reservada para la variable `persona`, sobreescribiendo impunemente otras variables aledañas.

Luego de que el primer usuario ingresa su nombre, la memoria quedaría así:



El carácter 'o' ha invadido el primer byte de la variable `edad` y el carácter de terminación, '\0', ha hecho lo propio con el segundo. El compilador no advierte este potencial error cuando verifica la sintaxis del programa ni tampoco el sistema arroja errores en tiempo de ejecución. La variable `edad` ha sido sobrescrita sin que nadie (excepto usted y yo) lo haya notado.

Ahora ya es fácil predecir lo que ocurrirá cuando el sistema imprima la edad del usuario. Por poseer una arquitectura *little-endian*, el sistema primero invertirá los cuatro bytes de la variable `edad` y luego imprimirá el valor codificado en:

00 00 00 6F = 00000000 00000000 00000000 0110 1111

Recuerde que el código ASCII de 'o' es 6F; y el carácter '\0' se representa con ocho bits nulos.

La larga cadena de bits mostrada es la representación del entero 111, la salida arrojada por el sistema.

Se puede inferir que cuando el segundo usuario ingrese su nombre, 'Alejandra', ahora será el carácter 'a' el que invada el primer byte de `num`. Lo que se imprimirá entonces será el valor decimal de 'a', cuyo código ASCII es 61_H=97.

No vaya a creer que el **desbordamiento de buffer** (*buffer overflow*) sólo ocurre cuando se trabaja con estructuras. En general, ocurre cada vez que se permite al usuario ingresar una cadena de texto que ha de ser almacenada en un arreglo de tamaño insuficiente. En este ejemplo hemos utilizado una estructura para poder realizar un «desbordamiento controlado». En general, es difícil predecir sobre qué variables podría desbordarse un texto ingresado desde el teclado.

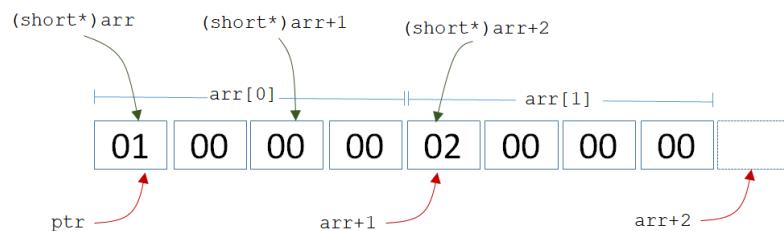
Para proteger nuestros programas contra esta silenciosa enfermedad se debe evitar leer cadenas con las funciones `gets` y `scanf`, y, en cambio, se debe utilizar la función `fgets` (declarada en `stdio.h`), que trunca las cadenas de texto que son más extensas que el arreglo donde se espera almacenarlas.

4. Determine manualmente cuál será la salida del siguiente programa cuando se ejecute sobre una arquitectura *little-endian*.

```
#include <stdio.h>
int main(void) {
    int arr[] = {1, 2};
    for(int i=0; i<3; i++)
        printf("%d", *((short*)arr + i));
}
```

Solución:

Se observa que `arr` es un arreglo de 2 elementos que ocupan 4 bytes cada uno. La representación del arreglo sobre una arquitectura *little-endian* sería de la siguiente forma:



Dentro del bucle `for` se imprimirán tres números enteros en base decimal (`%d`). Estos se encuentran en las siguientes direcciones:

`(short*)arr + i, i=0, 1, 2`

Note que debido a la conversión de tipos, la dirección `(short*)arr` no es de tipo `int*` sino de tipo `short*`. Siendo así, las direcciones `(short*)arr`, `(short*)arr+1` y `(short*)arr+2` se refieren a la 1^{ra}, 3^{ra} y 5^{ta} celdas mostradas en el gráfico anterior. Esto es lo que indica la aritmética de punteros de tipo `short*`.

De lo anterior, los tres valores a imprimir serán aquellos enteros cortos que comienzan en la 1^{ra}, 3^{ra} y 5^{ta} celda; o sea, los enteros 01 00, 00 00 y 02 00. En una arquitectura *little-endian*, estas son las representaciones internas de 1, 0 y 2, respectivamente, que son las salidas del programa.

PROBLEMAS PROPUESTOS

1. Escriba un programa que solicite el ingreso de un número flotante y que muestre los cinco siguientes flotantes que pueden ser guardados con exactitud, sin perdida de bits, en el formato IEEE754. Se sugiere utilizar uniones. Una posible interfaz sería como sigue:

```
>> Ingrese un flotante: 1984
1984.00000000000000000000000000000000: 0|10001001|11110000000000000000000000000000
```

Los cinco flotantes siguientes son:

```
1984.00012207031250000000000000: 0|10001001|11110000000000000000000000000001
1984.000244140625000000000000: 0|10001001|111100000000000000000000000000010
1984.000366210937500000000000: 0|10001001|111100000000000000000000000000011
1984.000488281250000000000000: 0|10001001|1111000000000000000000000000000100
1984.00061035156250000000000000: 0|10001001|1111000000000000000000000000000101
```

```
>> Ingrese un flotante: 1
```

```
1.00000000000000000000000000000000: 0|01111111|00000000000000000000000000000000
```

Los cinco flotantes siguientes son:

```
1.000000119209289550781250: 0|01111111|00000000000000000000000000000001
1.000000238418579101562500: 0|01111111|000000000000000000000000000000010
1.000000357627868652343750: 0|01111111|000000000000000000000000000000011
1.000000476837158203125000: 0|01111111|0000000000000000000000000000000100
1.000000596046447753906250: 0|01111111|0000000000000000000000000000000101
```

```
>> Ingrese un flotante: 0.5
```

```
0.50000000000000000000000000000000: 0|01111110|00000000000000000000000000000000
```

Los cinco flotantes siguientes son:

```
0.500000059604644775390625: 0|01111110|00000000000000000000000000000001
0.500000119209289550781250: 0|01111110|000000000000000000000000000000010
0.500000178813934326171875: 0|01111110|000000000000000000000000000000011
0.500000238418579101562500: 0|01111110|0000000000000000000000000000000100
0.500000298023223876953125: 0|01111110|0000000000000000000000000000000101
```

2. Dada la enumeración enum tipo_sangre {A=-182, B, AB, O}. Diga cómo se almacenará la variable enum tipo_sangre var = AB en una arquitectura *little-endian*.

3. Determine la salida del programa mostrado al ser ejecutado sobre una arquitectura *little-endian*:

```
#include <stdio.h>
int main(void) {
    int example[5] = {25,50,75,100};
    int *ptr = example;
    printf("%d",*((int*)((char*)ptr+4)));
}
```

4. Escriba un programa que imprima la representación, en *little-endian* y *big-endian*, de un arreglo de enteros cortos (*short*), cuyos valores son ingresados por el usuario.
5. **Interpretando una matriz como si fuese una estructura.** ¿Cuál será la salida del siguiente programa al ser ejecutado sobre una arquitectura *little-endian*? Asuma que la estructura *punto* no posee huecos en su representación interna.

```
#include <stdio.h>

struct punto { short x; short y; };

int main(void) {
    char M[2][2] = {0, 1, 2, 3};
    printf("%d %d", (*((struct punto*) &M)).x, (*((struct punto*) &M)).y);
}
```

CAPÍTULO

6

PUNTEROS A DATOS AGREGADOS

Anteriormente estudiamos cómo declarar punteros a datos atómicos. En este capítulo veremos como declarar punteros a datos agregados, p.ej. arreglos, matrices y estructuras de datos.

PUNTEROS A ARREGLOS

No es lo mismo un arreglo de punteros que un puntero a un arreglo. El primero es una colección de direcciones; el último contiene una sola dirección.

Los punteros a arreglos se definen con la siguiente sintaxis:

```
short array[3] = {1,2,3};  
short (*ptr)[3];  
ptr = &array;
```

En la primera línea se define un arreglo `array` de tres enteros cortos.

En la segunda línea se declara `ptr` como un puntero a un arreglo de 3 enteros cortos. Formalmente, el tipo de `ptr` es `short (*) [3]`.

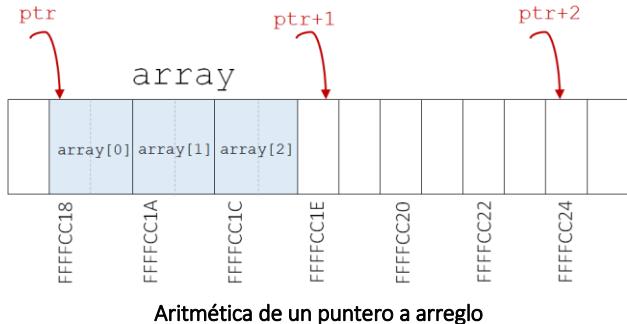
En la tercera línea se asigna `ptr` para que apunte al arreglo `array`, cuya dirección es especificada como `&array`. Es importante mencionar que el identificador `array` en la expresión `&array` no será interpretado como la dirección de su primer elemento, `&array[0]`. Recuerde que el nombre de un arreglo no decae cuando es precedido por el operador de dirección `&`.

La asignación `ptr = &array` habría arrojado un error de tipos incompatibles si `array` hubiese tenido una longitud distinta de 3 o si hubiese guardado elementos de un tipo distinto de `short`.

La dirección de un arreglo, `&array`, es exactamente igual que la dirección de su primer elemento, `&array[0]`. Sin embargo, la aritmética de punteros y la desreferencia siguen distintas reglas.

Con respecto a la aritmética de punteros, la dirección `ptr+1` es la dirección que se encuentra todo un arreglo más allá, más a la derecha de la dirección `ptr`. En nuestro caso, `ptr+1` apunta 6 bytes a la derecha de `ptr` porque ese, 6 bytes, es el tamaño del arreglo apuntado por `ptr`.

```
short (*ptr) [3] = &array;
```



Para hacer referencia a los elementos del arreglo `array` puede utilizarse `(*ptr) [0]`, `(*ptr) [1]` y `(*ptr) [2]`. Si se omitieran los paréntesis, los corchetes tendrían prioridad sobre el operador de desreferencia, `*`, y se obtendrían resultados inesperados.

También es posible declarar un puntero a un arreglo de tamaño arbitrario. Esto se logra haciendo:

```
short array[3] = {1, 2, 3};  
short (*ptr) [];  
//ptr apunta a un arreglo de tamaño  
//arbitrario  
ptr = &array;
```

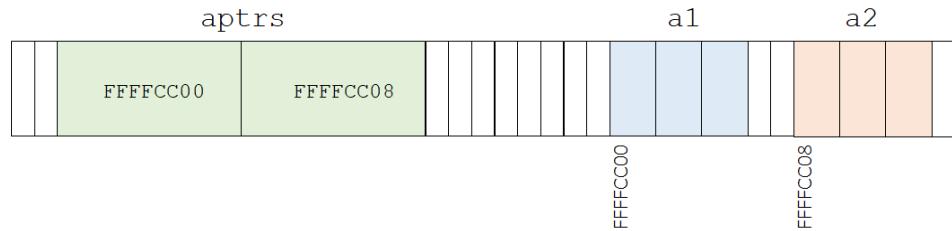
Pero en este caso no se podría recuperar el tamaño del arreglo apuntado por `ptr` con `sizeof(*ptr)`. El compilador nos recordará que es imposible recuperar el tamaño de un **tipo de dato incompleto**, `short (*) []`. Otro problema con los tipos de dato incompletos aparece cuando deseamos utilizar aritmética de punteros. La expresión `ptr+1` nuevamente genera un error en tiempo de compilación. El compilador necesita conocer el tamaño del dato apuntado por `ptr`, algo que no se puede obtener de la declaración de `ptr` como `short (*) []`.

ARREGLOS DE PUNTEROS A ARREGLOS

Uno podría agrupar varios punteros a arreglos en un mismo arreglo con la siguiente sintaxis:

```
short a1[3] = {1, 2, 3};  
short a2[3] = {4, 5, 6};  
short (*aptrs[2])[3] = {&a1, &a2};
```

Las expresiones `&a1` y `&a2` representan direcciones de arreglos, cada uno compuesto de tres enteros cortos. El arreglo `aptrs`, que es inicializado con `{&a1, &a2}`, es un arreglo que contiene dos punteros de tipo `short (*) [3]`. El tipo del arreglo `aptrs` es `short (*[2]) [3]`.



Tal como se observa en el gráfico anterior, los valores `aptrs[0]` y `&a1` son iguales; ambos se refieren al byte inicial del arreglo `a1`. Lo mismo ocurre con `aptrs[1]` y `&a2`.

El tamaño de `aptrs`, i.e. `sizeof(aptrs)`, es 16 bytes, el espacio de memoria que ocupan dos punteros. El tamaño del arreglo `a1` (6 bytes) puede obtenerse con `sizeof(*aptrs[0])` y, obviamente, también con `sizeof(a1)`.

Para acceder a los elementos de `a1` y `a2` a través de los punteros almacenados en `aptrs` puede utilizarse el siguiente código:

```
for(int i=0; i<2; i++)
    for(int j=0; j<3; j++)
        printf("%d", (*aptrs[i])[j]);
```

En la última línea, la expresión `*aptrs[i]` se refiere al `i`-ésimo arreglo cuya dirección está almacenada en `aptrs`. La expresión `(*aptrs[i])[j]` se refiere al `j`-ésimo elemento del `i`-ésimo arreglo cuya dirección está almacenada en `aptrs`. El uso de paréntesis es necesario pues, de lo contrario, los dos pares de corchetes, `[i] [j]`, tendrían mayor precedencia sobre el operador de desreferencia, `*`, lo que conduciría a resultados inesperados.

Terminamos mostrando la sintaxis para declarar arreglos de punteros a arreglos de tamaño arbitrario.

```
short a1[2] = {1, 2};
short a2[3] = {3, 4, 5};
short a2[4] = {6, 7, 8, 9};
short (*aptrs[])[] = {&a1, &a2, &a3};
```

Aunque el tamaño de `aptrs` puede ser obtenido con `sizeof(aptrs)`, los tamaños de los arreglos `a1`, `a2` y `a3` no pueden ser recuperados desde `aptrs`. Una expresión como `sizeof(*aptrs[1])` no retornará el tamaño del arreglo `a2` sino un error que nos recuerda que el compilador no puede deducir el tamaño de un tipo incompleto.

PUNTEROS A MATRICES

Un puntero a una matriz se define con la siguiente sintaxis:

```
int M[3][4];
int (*ptr)[3][4] = &M;
```

El puntero `ptr` está siendo asignado con la dirección de la matriz `M`. Si esta matriz no hubiese tenido dimensión 3×4 o si sus elementos hubieran sido de un tipo distinto a `int`, el compilador habría arrojado una advertencia de tipos de punteros incompatibles.

El (i, j) -ésimo elemento de `M` se puede obtener con `(*ptr)[i][j]`; el tamaño en bytes de `M`, con `sizeof(*ptr)`; el tamaño en bytes de la primera fila, `M[0]`, con `sizeof((*ptr)[0])`; el tamaño en bytes de un elemento, con `sizeof((*ptr)[0][0])`.

La expresión `ptr+1` se refiere a la dirección que está 48 bytes (el tamaño del tipo `int[3][4]`) más allá, más a la derecha de la dirección `ptr`.

También es posible definir `ptr` como un puntero a matrices de 4 columnas, sin especificar su número de filas. Esto se haría con el siguiente código:

```
int M[3][4], P[2][4];
int (*ptr)[][4];
ptr = &M;
ptr = &P;
```

Ahora `ptr` puede apuntar a matrices de 4 columnas y cualquier número de filas. Aunque nunca se podrá obtener el tamaño de `M` o `P` con `sizeof(*ptr)` a causa de la declaración incompleta del puntero `ptr`.

Nota:

Una asignación como `int (*ptr)[3][4] = M;` ocasiona un error de compilación por tipos de punteros incompatibles. Esto se debe a que, por decimaltamiento, el nombre de la matriz `M` se interpreta como la dirección de su primer elemento, i.e. de su primera fila, `&M[0]`. Si desea asignar `M`, debería hacerlo de la siguiente forma: `int (*ptr)[4] = M;` es decir, usando un puntero a arreglo.

ARREGLOS DE PUNTEROS A MATRICES

Un arreglo de punteros a matrices de 3×4 se define con la siguiente sintaxis:

```
int M[3][4], P[3][4];
int (*aptrs[2])[3][4] = {&M, &P};
```

No haremos comentarios del código anterior. Se deja el análisis al lector.

PUNTEROS A ESTRUCTURAS

Un puntero a una estructura se define del siguiente modo:

```
typedef struct Tupla {float x; float y;} Punto2D;
Punto2D mipunto = (Punto2D) {3.0, 5.0};
Punto2D *ptr;
ptr = &mipunto;
```

En la primera línea del código anterior la estructura `Tupla` es declarada y rebautizada con el alias `Punto2D`. Este último nombre ahora puede ser convenientemente utilizado en el resto del programa en lugar del extenso y formal `struct Tupla`. Cada variable del tipo `Punto2D` poseerá dos atributos de tipo flotante, `float`, que almacenarán las componentes (`x`, `y`) de un punto bidimensional.

La variable `mipunto` es inicializada con `x=3.0, y=5.0` en la segunda línea del código anterior.

La tercera línea contiene la declaración de un puntero a la estructura `Punto2D`. Hubiese sido lo mismo si se usaba la declaración `struct Tupla *ptr`.

En la última línea se utiliza el operador `&` para obtener la dirección de `mipunto` y asignarla a `ptr`.

Existen dos formas de acceder a los atributos de `mipunto` a través del puntero `ptr`.

- La primera es utilizando el **operador punto**. Esto nos permite acceder a los atributos de la estructura apuntada por `ptr` con las siguientes expresiones:

$$(*ptr).x \quad y \quad (*ptr).y$$

El uso de paréntesis es obligatorio para evitar la mayor precedencia que tendría el operador punto sobre el operador de desreferencia.

- La segunda forma es usando el **operador flecha**, `->`. Aquí no hay necesidad de desreferenciar el puntero `ptr`. Se puede acceder directamente a los atributos de `mipunto` haciendo:

$$ptr->x \quad y \quad ptr->y$$

En cuanto a la aritmética de punteros, la expresión `ptr+1` es la dirección que se encuentra N bytes a la derecha de `ptr`, siendo N el tamaño en bytes de la estructura, `sizeof(Punto2D)`.

PUNTEROS A ARREGLOS DE ESTRUCTURAS

Asumiendo la misma estructura `Tupla` de alias `Punto2D` definida anteriormente, un puntero `ptr` a un arreglo de estructuras se definiría así:

```
Punto2D puntos[100];
Punto2D (*ptr)[100] = &puntos;
```

Una manera de navegar el arreglo apuntado por `ptr` sería del siguiente modo:

```
for(int i=0; i<100; i++)
    printf("%f, %f\n", (*ptr)[i].x, (*ptr)[i].y);
```

La expresión `*ptr` hace referencia a todo el arreglo `puntos`, cuyo i -ésimo elemento será pues `(*ptr)[i]`. Cada uno de estos elementos no es un dato atómico sino una estructura con dos atributos, los cuales pueden ser referenciados usando `(*ptr)[i].x` y `(*ptr)[i].y`.

Otra manera de navegar el arreglo `puntos` sería ignorando el puntero `ptr` y accediendo a los elementos por medio de desplazamientos (*offsets*). Dicho código sería así:

```
for(int i=0; i<100; i++)
    printf((puntos+i)->x, (puntos+i)->y)
```

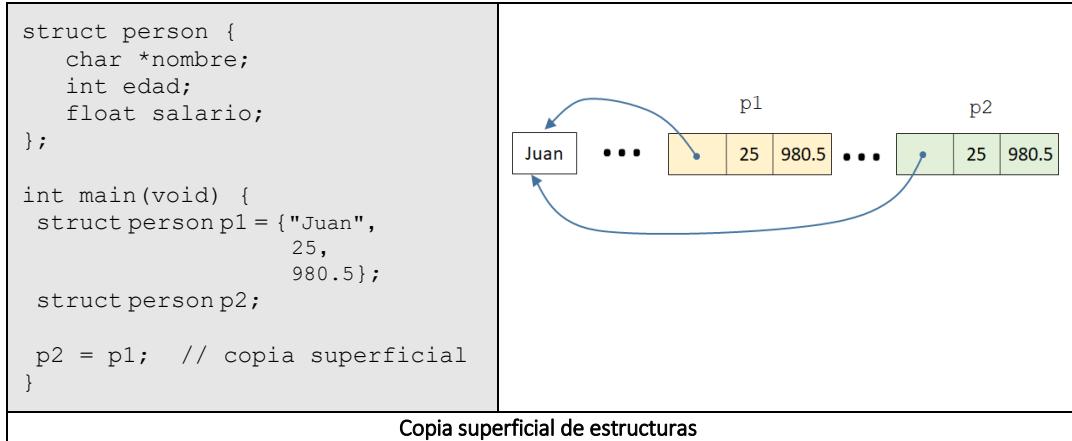
Conforme se incrementa el valor de `i` dentro del bucle `for` la expresión `puntos+i` va apuntando cada vez más hacia la derecha, hacia la siguiente estructura a partir de `puntos[0]`. En otras palabras, `puntos+i` es la dirección de la i -ésima estructura del arreglo `puntos`. Por tratarse de la dirección de una estructura, es posible acceder a sus atributos por medio del operador flecha, con `(puntos+i)->x` y `(puntos+i)->y`.

COPIA PROFUNDA Y COPIA SUPERFICIAL DE ESTRUCTURAS

Cada vez que hacemos una asignación de estructuras de la forma `A=B` estamos realizando una **copia superficial** (*shallow copy*) de la estructura `B` en la estructura `A`. Todos los bytes que se utilizan para codificar la estructura `B` serán replicados en el espacio de memoria reservado para `A`. Esto trae una consecuencia. Si las estructuras tuviesen algún puntero como atributo, dichos punteros, tanto en `A` como en `B`, quedarán apuntando a las mismas direcciones luego de `A=B`.

Abajo se muestra la implementación de una copia superficial y un diagrama con una posible configuración de la memoria luego de la copia.

En el código de la izquierda, la asignación `p2 = p1` hace que los punteros de ambas estructuras terminen apuntando al mismo dato. Un cambio en el dato común afectaría a todas las estructuras que lo apuntan.

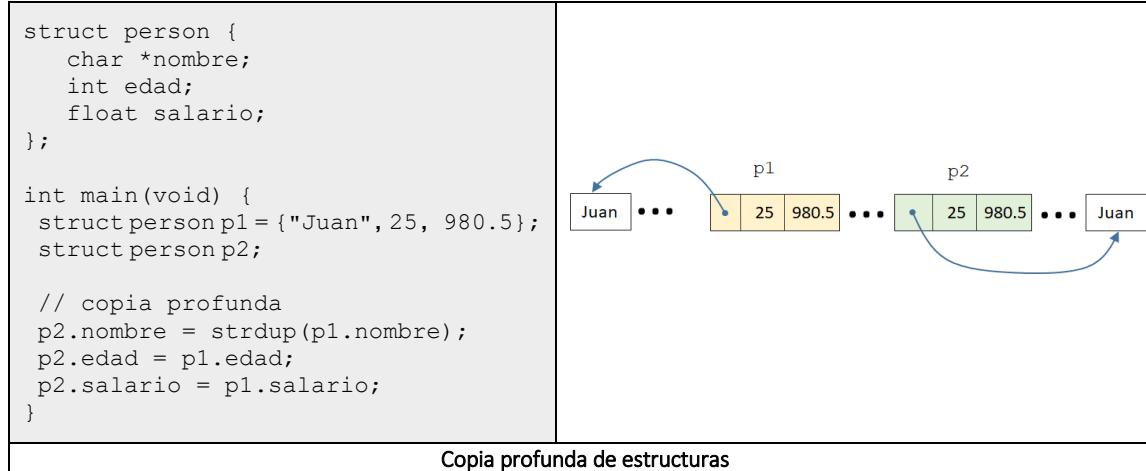


También se habría realizado una copia superficial si en lugar de `p2 = p1` se hubiese hecho:

```
memcpy(&p2, &p1, sizeof(struct person))
```

Lo anterior indica que se copien tantos bytes de la dirección `&p1` a la dirección `&p2`. El prototipo de `memcpy` se encuentra en `string.h`.

Una forma distinta de copiar estructuras es la **copia profunda** (*deep copy*), que se ilustra abajo. Con este método ya no se copian los punteros sino los datos apuntados por los punteros.



La función `strdup` (con prototipo en `string.h`) crea una copia del dato apuntado por el puntero `nombre` en otro lugar de la memoria. Tal como se muestra en el gráfico, ahora cada estructura apunta a sus propios datos.

Cuando la estructura que se desea copiar no contiene ningún puntero como atributo es indiferente hacer una copia profunda o una superficial. De hecho, estos conceptos ni siquiera tendrían sentido. Bastaría con hablar simplemente de copia.

PROBLEMAS RESUELTOS

1. Complete las siguientes declaraciones de modo que sean sintácticamente válidas. No utilice punteros genéricos, void*.

```
int *px, *py, M1[3][4], M2[3][4];  
  
[ ] = {&M1[1][1], &M2[2][2]};  
[ ] = {&M1, &M2};  
[ ] = {&px, &py};  
[ ] = {"Juan", "Ernesto", "Axl"};  
[ ] = {M1[0], M2[0]};  
[ ] = {&M1[0], &M2[0]};
```

Solución:

Analizaremos cada línea por separado.

- Dado que M1[1][1] y M2[2][2] son enteros, entonces el arreglo {&M1[1][1], &M2[2][2]} contiene direcciones de enteros. Por lo tanto debería ser definido como un arreglo de dos punteros a enteros, int *array1[2].
- Como &M1 es la dirección de una matriz, entonces {&M1, &M2} es un arreglo de direcciones a matrices de enteros (de 3x4). Esto puede definirse como int (*array2[2])[3][4].
- Dado que px y py son punteros a enteros, {&px, &py} es un arreglo con dos direcciones a punteros, i.e. un arreglo de dos punteros a punteros, int **array3[2].
- Un texto encomillado puede ser declarado como tipo char*. Luego, el arreglo de nombres mostrado arriba puede ser asignado a char* array4[3].
- M1[0] es el nombre de la primera fila de M1; o sea, el nombre de un arreglo. Por decaimiento, este nombre de arreglo es interpretado como la dirección de su primer elemento, &M1[0][0]. Por lo tanto, {M1[0], M2[0]} puede ser definido como int* array5[2], un arreglo de dos punteros a enteros.
- &M1[0] es la dirección de la primera fila de M1. A diferencia del caso anterior, M1[0] ya no decae porque ahora está precedido por el operador &. El arreglo {&M1[0], &M2[0]} contiene dos direcciones de filas (i.e. direcciones a arreglos de 4 elementos). Dicho arreglo puede ser asignado a int (*array6[2])[4], un arreglo de dos punteros a arreglos de 4 enteros.

2. **Decaimiento de matrices.** Determine la salida del siguiente programa:

```
int main(void) {
    int A[3][4] = {{1}, {2, 3}, {4, 5, 6}};

    printf("%d ", *(A[1] + 1));
    printf("%d ", *(*(A + 2) + 1));
    printf("%d ", *(A[2] - 4));
    printf("%d ", (*(A + 1))[1]);
}
```

Solución:

`A` es el nombre de una matriz; `A[i]`, $i=0, 1, 2, \dots$, son referencias a las filas de dicha matriz.

- `A[1]` se refiere a la segunda fila de `A`; es decir, al arreglo `{2, 3}`. Por decaimiento, el arreglo `A[1]` se interpreta como la dirección de su primer elemento; es decir, como `&A[1][0]`, la dirección donde se encuentra el número 2. Siendo así, `A[1]+1` sería la dirección que se encuentra 4 bytes a la derecha de `&A[1][0]`. El dato contenido en esta dirección, `* (A[1]+1)`, es 3. Eso es lo que imprime el primer `printf`.
- En cuanto a la segunda sentencia `printf`, por decaimiento, el nombre de la matriz `A` se interpreta como la dirección de su primer elemento. Dado que los elementos de una matriz son sus filas, `A` sería la dirección de la primera fila y `A+2` sería la dirección de la tercera fila. Entonces `* (A+2)` ya no es una dirección, sino una referencia a la tercera fila, el nombre de la tercera fila, y, como tal, decae a la dirección del primer elemento de la tercera fila. O sea, `* (A+2)` es la dirección del 4, y `* (A+2)+1` es la dirección del 5. Este valor es lo que imprime el segundo `printf` luego de desreferenciar toda la expresión anterior con `* (* (A+2)+1)`.

El análisis de las dos últimas líneas del código anterior se deja al lector.

3. Determine la salida del siguiente programa bajo una arquitectura *little-endian*:

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

int main(void) {
    float x;
    memset((void*)&x+3, 128, 1);
    memset(&x, 0, 3);
    printf("%f", x);
}
```

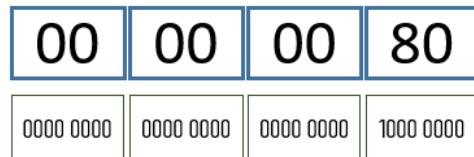
Solución:

La función `memset(ptr, c, N)` con prototipo en `string.h` permite llenar las `N` celdas de memoria que se encuentran después de la dirección `ptr` con el valor `c` (tipo `unsigned char`). Dicho esto, según el código mostrado, los cuatro bytes reservados para la variable `float` serán llenados a través de las dos sentencias `memset`.

La primera sentencia `memset` llena el cuarto byte ocupado por la variable `x` con el valor 128. Observe que esta celda se encuentra tres celdas a la derecha de `&x` y, por lo tanto, su dirección se obtiene con `(void*) &x+3`. El molde `(void*)` es necesario para alterar la aritmética de punteros. Su omisión hubiera hecho que `&x+3` apunte al doceavo byte después de `&x`.

La segunda sentencia `memset` llena los tres primeros bytes reservados para `x`. Es decir, llena la celda ubicada en `&x`, la siguiente y la subsiguiente, cada una con el valor de 0.

A sabiendas que 128 y 0 se codifican como 1000 0000 y 0000 0000, respectivamente, el espacio reservado para `x` quedaría de la siguiente forma:



Ahora sólo falta deducir que imprimirá el programa. Aquí recién se necesita conocer el *endianness* de la computadora. En una *little-endian*, el programa imprimirá el valor codificado en la secuencia 80 00 00 00.

Como ya se dijo en el capítulo de *Representación Interna de Datos Atómicos*, cuando un número flotante está representado por una cadena de bits nulos, excepto por el bit de signo, su valor será considerado como -0 . 000000, el cero negativo. Esa será la salida del programa.

PROBLEMAS PROPUESTOS

1. Los dos programas siguientes imprimen las diez copias que se hace de una cadena de texto original. Luego de modificar el texto original, las diez copias vuelven a imprimirse obteniéndose diferentes salidas para cada programa. Explique la razón de esta diferencia.

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

int main(void) {
    char original[100] = "Un texto";
    char *copias[10];

    for(int i=0; i<10; i++)
        copias[i] = original;

    for(int i=0; i<10; i++)
        printf("%s\n", copias[i]);

    strcpy(original, "Otro texto");

    for(int i=0; i<10; i++)
        printf("%s\n", copias[i]);
}
```

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

int main(void) {
    char *original = "Un texto";
    char *copias[10];

    for(int i=0; i<10; i++)
        copias[i] = original;

    for(int i=0; i<10; i++)
        printf("%s\n", copias[i]);

    original = "Otro texto";

    for(int i=0; i<10; i++)
        printf("%s\n", copias[i]);
}
```

2. Analice los siguientes programas y explique por qué producen diferentes salidas.

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

int main(void) {
    char original[100] = "Un texto";
    char *copias[10];

    for(int i=0; i<10; i++)
        copias[i] = original;

    for(int i=0; i<10; i++)
        printf("%s\n", copias[i]);

    strcpy(original, "Otro texto");

    for(int i=0; i<10; i++)
        printf("%s\n", copias[i]);
}
```

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

int main(void) {
    char original[100] = "Un texto";
    char *copias[10];

    for(int i=0; i<10; i++)
        copias[i] = strdup(original);

    for(int i=0; i<10; i++)
        printf("%s\n", copias[i]);

    strcpy(original, "Otro texto");

    for(int i=0; i<10; i++)
        printf("%s\n", copias[i]);
}
```

3. Determine la salida del siguiente programa. Asuma una arquitectura *little-endian*

```
#include <stdio.h>

int main(void) {
    int (*p1)[3];
    int (*p2[4])[3];
    int (*p3)[4][3];
    int * const p4[4][3];

    printf("%lu\n", sizeof(p1));
    printf("%lu\n", sizeof(p2));
    printf("%lu\n", sizeof(p3));
    printf("%lu\n", sizeof(p4));
}
```

CAPÍTULO

7

FUNCIONES

Cuando los programas crecen en tamaño y complejidad se hace cada vez más difícil escribirlos dentro una sola función (`main`), como si fueran piezas monolíticas de código. Por un lado, la enorme cantidad de instrucciones y variables revoloteadas hacen que sea difícil su lectura, comprensión y mantenimiento. Por otro lado, la implementación se vuelve ineficiente. Es probable encontrarse con fragmentos de código que se repiten varias veces dentro del programa, p.ej. sumar los datos de un arreglo o resolver una ecuación de segundo grado. Estos problemas pueden ser, sino completamente resueltos, por lo menos atenuados mediante el uso de funciones.

Una función es un conjunto de instrucciones agrupadas bajo un nombre. Son definidas para realizar una tarea específica. Cada función incluye sus propias variables e instrucciones. Esto permite distribuir las variables de un programa en distintas piezas de código en lugar de tenerlas todas juntas, dentro de una única función principal. El resultado es un código mejor organizado y más legible en comparación con el **espagueti de código** de los programas monolíticos.

Otra característica de las funciones es que son reutilizables. Su funcionalidad puede ser invocada varias veces, en distintos puntos del programa. Así, aquellos fragmentos de código que serían repetitivos dentro de un programa monolítico pueden ser agrupados dentro de una función, que luego puede ser invocada cuantas veces sea necesario. De esta manera se reduce la **redundancia de código**.

Un programa en C es un conjunto de funciones que debe incluir obligatoriamente una función principal caracterizada por el nombre `main`. La existencia de una función principal hace que se establezca una jerarquía en las llamadas a funciones: La función `main` puede invocar otras funciones quienes, a su vez, pueden invocar otras más. Cada vez que una función invoca a otra, el flujo de control pasa de la función que invoca a la función que es invocada.

DEFINICIÓN DE UNA FUNCIÓN

La sintaxis para definir una función es la siguiente:

```
tipo_retorno nombre_funcion (lista_de_parámetros)
{
    cuerpo_funcion;
}
```

La primera línea de la definición se denomina **cabecera de la función**. Consta de tres partes:

- El **nombre de función** es un identificador que permite invocar a la función, pedirle que ejecute la tarea para la que fue definida.
- La **lista de parámetros** es un conjunto de variables que permite que la función pueda recibir datos desde afuera, desde la función llamadora, al momento de su invocación.
- El **tipo de retorno** indica el tipo de dato que se devolverá a la función llamadora.

Debajo de la cabecera debe definirse un bloque de código denominado **cuerpo de la función**. El cuerpo está compuesto por un conjunto de instrucciones que definen el comportamiento de la función, lo que la función debe hacer cada vez que sea invocada.

INVOCACIÓN DE UNA FUNCIÓN

Para ejecutar el código de una función se necesita llamarla o invocarla haciendo referencia a su nombre. Es muy posible que la función llamada requiera una serie de datos para realizar sus cálculos. Si así fuera, la invocación podría incluir esta lista de datos. También es posible que la función llamada necesite retornar algún resultado a la función llamadora. En este caso, dicho resultado podría ser devuelto y capturado en una variable, por citar uno de los posibles casos.

```
#include <stdio.h>

int suma (int a, int b)
{
    int S;
    S = a+b;
    return S;
}

int main(void) {
    int result;

    result = suma(5,3);
    printf("5+3=%d", result); // imprime 8

    result = suma(9,-2);
    printf("9+-2=%d", result); // imprime 7
}

milfun.c
```

Para ilustrar algunos de los conceptos relacionados con las funciones comentaremos el programa `milfun.c`, que está compuesto por dos funciones: `suma` y `main`.

Empecemos por la función principal, `main`. Desde aquí se realizan dos llamadas a `suma`. En ambos casos `main` vendría ser la función llamadora; y `suma`, la función llamada. Note que en cada invocación `suma` recibe diferentes datos de entrada. En el primer caso, 5 y 3; en el segundo, 9 y -2. Los resultados de cada llamada, 8 y 7, son capturados en la variable `result`, que es impresa en pantalla inmediatamente después de cada invocación.

Por otro lado, la función `suma` ha sido definida para recibir dos datos enteros desde la función llamadora. Esto se deduce de la lista de parámetros (`int a, int b`) incluidos en su cabecera. Las funciones que no necesitan datos externos para realizar sus cálculos deben incluir (`void`) como lista de parámetros. La función `suma` también retornará un dato entero a la función llamadora. Esto se deduce del tipo de retorno `int` usado en la cabecera de `suma`. Las funciones que no necesitan retornar ningún valor a la función llamadora deben declararse con tipo de retorno `void`.

Dentro de una función uno podría declarar variables como `s`. Dichas variables se denominan **variables locales** porque sólo pueden ser referenciadas desde dentro de una función; afuera serían irreconocibles. El valor retornado por `suma` se especifica con `return s`. Del código puede deducirse que lo que se retorna es el resultado de la suma `a+b`, acumulado localmente en la variable `s`. Las funciones con tipo de retorno `void` no necesitan incluir una sentencia `return`.

Los **argumentos** de una función son los datos pasados desde la función llamadora; los **parámetros** son las variables que se usan para capturar estos datos. En el programa mostrado, las variables `a` y `b` son los parámetros de `suma`; los datos 5 y 3 son los argumentos usados en la primera invocación; y 9 y -2, los argumentos usados en la segunda. Los argumentos no necesariamente tienen que ser constantes, también pueden ser variables o incluso expresiones, p.ej. `suma(x, 4*y+5*x)`.

Debemos confesar que la implementación mostrada de `suma` es mucho más compleja de lo necesario. También pudo haberse definido en una sola línea haciendo:

```
int suma(int a, int b) { return a+b; }
```

Análogamente, el cuerpo de la función principal pudo definirse más brevemente, en dos líneas:

```
printf("%d", suma(5,3));
printf("%d", suma(9,-2));
```

La expresión `suma(5,3)` es tratada como si fuera un número entero, `%d`, pues ese es el tipo de retorno de `suma`.

PASO POR VALOR Y PASO POR REFERENCIA

Cada vez que pasamos una variable como argumento a una función debemos tener muy claro qué se espera recibir dentro de la función: (1) el valor de la variable, o (2) la variable en sí. Para el primer caso debemos implementar el paso de argumento por valor; para el segundo, el paso por referencia.

Los siguientes ejemplos muestran las dos formas posibles de pasar la variable `N` a la función `add100`:

<pre>void add100(int a) { a = a + 100; } int main(void){ int N = 5; printf("%d", N); // imprime 5 add100(N); printf("%d", N); // imprime 5 }</pre>	<pre>void add100(int *pa) { *pa = *pa + 100; } int main(void){ int N = 5; printf("%d", N); // imprime 5 add100(&N); printf("%d", N); // imprime 105 }</pre>
Paso por valor	Paso por referencia

Empecemos describiendo el código de la izquierda. Aquí la invocación `add100 (N)`, dentro de la función `main`, indica que lo que se pasa a `add100` será el dato contenido en la variable `N`, no la variable en sí. Dicho dato podrá ser utilizado de cualquier forma imaginable dentro de la función `add100` sin que esto afecte a la variable `N` que se encuentra definida afuera, en `main`. La variable `N` no se verá afectada por la llamada a `add100` y mantendrá su valor original, 5, después de esta llamada. Aunque invoquemos mil veces a `add100` el valor de `N` será siempre el mismo. La variable `N` nunca ingresa a la función `add100` y, por lo tanto, su valor no puede ser cambiado desde allí adentro. En nuestro ejemplo, dado que `N=5`, la invocación `add100 (N)` equivale a la llamada `add100 (5)`. El valor 5 se copia en el parámetro `a` de la función `add100`, y es este parámetro el que se modifica dentro de `add100`, no `N`.

Ahora pasemos a comentar el código de la derecha. Aquí la variable `N` ha sido pasada por referencia a `add100`. Esto significa que el valor de `N` sí puede ser cambiado dentro de `add100`, como de hecho ha ocurrido, ha cambiado de 5 a 105. El artificio que se requiere para implementar el paso por referencia consiste en pasar la dirección de la variable que se desea modificar. Cada vez que se ejecuta la llamada `add100 (&N)`, la dirección `&N` se guarda en el parámetro `pa`; así evitamos que se pierda contacto con la variable `N` que, aunque se encuentre afuera, en otra función (`main`), siempre puede ser referenciada mediante `*pa`. En resumen, el valor de `N` siempre puede ser modificado desde dentro de `add100` desreferenciando la dirección `&N` almacenada en el parámetro `pa`.

Nota

Cuando deseemos que el valor de una variable `var` pueda ser modificado desde adentro de una función `foo`, no deberemos pasar la variable como argumento, p.ej. `foo (var)`, sino la dirección de la variable, p.ej. `foo (&var)`. En el primer caso sólo estaríamos pasando el dato contenido en `var`; en el segundo, su dirección, `&var`, que siempre puede ser usada como una ruta directa hacia `var`.

PASO DE ARREGLOS A FUNCIONES

Cada vez que usamos el nombre de un arreglo como argumento en una llamada a función estamos haciendo una llamada por referencia. En cambio, cuando pasamos el nombre de una variable atómica o una estructura estamos utilizando el paso de argumento por valor.

Ilustramos ambas afirmaciones con el siguiente ejemplo:

```

void add100(int arr[], int N) {
    for(int i=0; i<N; i++)
        arr[i] = arr[i] + 100;
}

void imprime(int arr[], int N) {
    for(int i=0; i<N; i++)
        printf("%d ", arr[i]);
}

int main(void){
    int array[3] = {10, 20, 30};
    int size = 3;
    imprime(array, size); // imprime 10 20 30
    add100(array, size);
    imprime(array, size); // imprime 110 120 130
}

```

Paso de arreglos

Dentro del `main`, en la llamada `add100(array, size)`, el argumento `array` es pasado por referencia mientras que el argumento `size` es pasado por valor. Lo primero significa que las operaciones realizadas al interior de la función `add100` pueden modificar los datos del arreglo `array`. De hecho así ocurre. Cada elemento de `array` es añadido en 100 unidades dentro de `add100`. Lo segundo significa que la variable `size` sólo puede ser leída pero no modificada desde dentro de `add100`. `array` es pasada por referencia porque es (por decreimiento) una dirección; `size`, en cambio, es un nombre de variable.

En cuanto a `add100`, también pudo usarse `void add100(int *arr, int N)` como cabecera y su funcionalidad habría sido la misma. En cualquier caso, es imposible recuperar el tamaño del arreglo `array` haciendo `sizeof(arr)/sizeof(int)` dentro de `add100`. La expresión `sizeof(arr)` devolverá 8, el tamaño en bytes de un puntero. Por ello, cada vez que pasamos arreglos a una función debemos pasar también el tamaño `N` de los mismos, sólo así podremos navegar todos sus elementos, desde el primero (subíndice 0) hasta el último (subíndice `N-1`). El único caso donde no es necesario pasar el tamaño de un arreglo a una función es cuando pasamos una cadena. En estos casos se puede aprovechar el hecho que las cadenas terminan con el carácter '\0'. Así, sin necesidad de conocer su tamaño, una cadena puede desde visitada desde su primer elemento (subíndice 0) hasta el último, aquel donde se cumpla que `array[i]=='\0'`.

Pasar arreglos por referencia es la forma correcta y eficiente de programar. Evita que el sistema tenga que copiar un montón de datos a las variables locales de la función invocada.

PASO DE MATRICES A FUNCIONES

Al igual que los arreglos, las matrices también son pasadas por referencia a una función. Esto implica que cada vez que hacemos una invocación de la forma `foo(M)` las alteraciones que sufre la matriz `M` dentro de `foo` serán preservadas luego del retorno a la función llamadora.

Discutamos el siguiente programa:

```
void mImprime (int N, int M[N] [N]) {
    for(int i=0; i<N; i++) {
        for(int j=0; j<N; j++)
            printf("%d ", M[i] [j]);
        printf("\n");
    }
}

void mIdentidad(int N, int M[N] [N]) {
    for(int i=0; i<N; i++)
        for(int j=0; j<N; j++)
            M[i] [j] = i==j?1:0;
}

int main(void) {
    int M[5][5];      // M tiene cualquier valor
    mIdentidad(5,M); // M se actualiza
    mImprime(5,M);   // Imprime M actualizada
}
```

En la primera línea del `main` la matriz `M` ha sido declarada pero no se le ha asignado valor a ninguno de sus 25 elementos. Estos recién serán asignados dentro de la función `mIdentidad`, en la segunda línea del `main`, con la invocación `mIdentidad(5,M)`. Finalmente, cuando se imprime la matriz `M`, con `mImprime(5,M)`, el sistema mostrará en pantalla la matriz `M` ya modificada.

La función `mIdentidad` recibe una matriz arbitraria `M` y la convierte en matriz identidad. Por definición, para que `M` sea una matriz identidad todos los elementos de su diagonal principal deberían ser unos; y el resto, ceros. O sea, `M[i] [j] = i==j?1:0`.

Aprovecharé la definición de `mIdentidad` para decir algunas cosas importantes sobre las funciones que reciben matrices. Primero, la cabecera de `mIdentidad` no puede definirse del siguiente modo:

```
void mIdentidad(int N, int M[] [])
```

Cuando se declara una matriz es obligatorio especificar, por lo menos, su número de columnas. Una declaración como:

```
void mIdentidad(int N, int M[] [5])
```

sí habría sido correcta y no habría causado problemas. Pero, ¿quién quiere ver ese 5 allí? No sólo me refiero a su desagradable efecto estético, sino también a que la función `mIdentidad`, así definida, solamente podría recibir matrices de 5 columnas, ni una más, ni una menos.

Dado que lo deseable es definir funciones que sean lo más genéricas posible, los parámetros utilizados para recibir matrices debería definirse como `int M[nfils][ncols]`; es decir, de manera general. Un parámetro definido así exigiría que `nfils` y `ncols` hayan sido previamente declaradas. Y así ocurre en una cabecera como la que se muestra abajo:

```
void foo(int nfils, int ncols, int M[nfils][ncols])
```

Volviendo al programa que estamos comentando, la función `mImprime` también podría haber causado cambios en la matriz `M`, pero no lo ha hecho. En ninguna parte de su código encontramos una asignación de la forma `M[i][j]=valor`, lo que habría alterado los elementos de `M`.

FUNCIONES QUE RETORNA PUNTEROS

Una función puede retornar direcciones de memoria siempre y cuando estas no apunten a variables locales definidas dentro de la misma. Por ejemplo, el siguiente código lanza una advertencia en la línea `return &var` de la función `foo`.

```
int* foo(void) {
    int var;
    return &var; // mal hecho
}

int main(void) {
    int* ptr = foo(); // ptr apunta a una zona
                      // de memoria ya liberada
}
```

No retornar direcciones de variables locales

Esta advertencia nos recuerda que una vez concluida la llamada a `foo` la variable `var` será liberada. El mismo espacio de memoria que ocupó `var` durante la ejecución de `foo` podría luego ser ocupado por otros datos. Por eso no es buena idea utilizar la dirección de `var` en el `main`. Uno podría, inocente o maliciosamente, desreferenciarla y escribir en celdas de memoria que ya están siendo usadas para otros propósitos.

Una forma correcta de retornar punteros es la que se muestra en la página siguiente.

Ahora es la función `posicion` la que retorna un puntero, `int*`. En mayor detalle, esta función recibe un arreglo `arr` de `N` elementos y retorna la dirección del elemento donde se almacena `valor`. Cuando fracasa la búsqueda retorna `NULL`.

Esta vez el compilador no lanza ninguna advertencia ya que en ningún momento `posicion` devuelve direcciones locales. Lo que retorna son direcciones `&arr[i]`, las cuales, incluso luego de concluida la llamada a `posicion(a, 7, 0)`, continúan tan «vigentes» como el arreglo `a` a cuyos elementos apuntan.

```
#include <stdio.h>

int* posicion(int arr[], int N, int valor) {
    for(int i=0; i<N; i++)
        if(arr[i] == valor)
            return &arr[i];
    return NULL;
}

int main(void) {
    int a[7] = {3, 4, 6, 0, 8, 2, 9};
    int *ptr = posicion(a, 7, 0);
    if(ptr!=NULL)
        printf("Hallado en a[%d]", ptr-a);
}
```

Retornando la dirección de un valor específico

Dentro del `main`, la dirección retornada desde `posicion` es asignada al puntero `ptr`, el cual se utiliza para imprimir el subíndice del elemento buscado, `ptr-a`. La salida del programa mostrado será: Hallado en a [3], refiriéndose al número 0.

PROTOTIPO DE FUNCIONES

Un **prototipo** es una declaración de función. Indica el nombre de la función, su tipo de retorno y los tipos de sus parámetros. No es una definición de función porque no describe el comportamiento de la misma, sólo su interface, su modo de intercambiar datos con la función llamadora.

Hasta el momento nunca habíamos tenido necesidad de este concepto. Siempre habíamos podido definir nuestras funciones antes de la función principal, `main`. Lamentablemente existen casos en los que esto no es posible. Por citar un ejemplo, podríamos tener dos funciones `A` y `B`, en cuyos códigos la función `A` invoca a `B` y `B` invoca a `A`. En este caso, ¿qué función deberíamos definir primero? Si definimos `A` primero, el compilador lanzará un error diciendo que `A` está invocando a una función que aún no ha sido declarada, `B`. Y una respuesta semejante obtendríamos si definimos primero `B`. Este espinoso caso se resuelve utilizando prototipos.

En el código mostrado abajo, nuestras hipotéticas funciones `A` y `B` han sido encarnadas en las funciones `par` e `impar`. Cada una de estas invoca a la otra. A causa de esta característica ninguna de las dos puede ser definida antes del `main` sin causar conflictos. Por ello el programa no define ninguna de las dos funciones antes del `main`, sólo las declara.

```

bool impar(int);    // declaración de impar
bool par(int);      // declaración de par

int main(void){
    printf("%d", par(5)); // imprime 0, o sea falso
}

bool impar(int N) { // definicion de impar
    if(N==0)
        return false;
    else
        return par(N-1);
}

bool par(int N) { // definicion de par
    if(N==0)
        return true;
    else
        return impar(N-1);
}

```

La declaración de `impar` es la sentencia: `bool impar(int);` la de `par` es `bool par(int)`. Una declaración se parece una cabecera de función; pero ni siquiera es eso. La cabecera debe incluir los nombres y tipos de los parámetros. La declaración sólo exige los tipos de estos.

Luego de las declaraciones de `par` e `impar` estas funciones ya pueden ser definidas, por lo general, debajo del `main`. Allí ya no importa qué función se define primero y cuál después. Sea cual fuere el orden, siempre que las definiciones se encuentren debajo de las declaraciones ya no se violará aquella regla que sólo permite invocar funciones previamente declaradas.

PROGRAMAS MULTIARCHIVO

Un segundo caso donde se necesita usar prototipos es cuando implementamos programas multiarchivo. A veces, por razones de organización, es deseable escribir las funciones de un programa en diferentes archivos. Al final siempre es posible integrar la funcionalidad dispersa en varios archivos fuentes en un único ejecutable. De hecho, esa es la tarea del enlazador (*linker*).

Discutamos el siguiente programa de dos archivos, cuya salida se muestra en los comentarios:

<pre> #include <stdio.h> int cuadrado(int); int cubo(int); int main(void) { int a = cuadrado(5); printf("a=%d\n", a); // 25 int b = cubo(5); printf("b=%d\n", b); //125 } </pre>	<pre> int cuadrado(int x) { return x*x; } int cubo(int x) { return x*cuadrado(x); } </pre>
<code>pcpal.c</code>	<code>funs1.c</code>

En un programa multiarchivo sólo uno puede contener la función `main` (`pcpal.c` en nuestro caso).

Tal como se observa, desde `pcpal.c` es posible invocar las funciones `cuadrado` y `cubo`, que están definidas en otro archivo, en `funs1.c`. Pero para ello el archivo `pcpal.c` debe incluir los prototipos de las funciones externas `cuadrado` y `cubo`. De lo contrario el compilador arrojaría un error diciendo que `main` está invocando funciones que no han sido previamente declaradas.

En este punto algún escéptico podría cuestionar: ¿Y por qué el compilador se quejaría cuando invocamos a `cuadrado` (asumiendo que omitimos su declaración en `main`), pero nunca se ha quejado de la función `printf`, por ejemplo, que ha sido utilizada sin problemas desde nuestro primer programa, a pesar de que nunca hayamos escrito su declaración? La respuesta es que `printf` sí ha sido declarada, quizás no explícitamente, por nosotros mismos; pero la declaración siempre ha estado presente dentro del archivo `stdio.h`. Puede usted mismo buscar este archivo en su sistema, abrirlo con un procesador de texto, y navegar hasta encontrar la declaración de `printf`, que ha estado siempre a la vista del compilador.

Para integrar los dos archivos anteriores en un único ejecutable debe hacerse:

```
gcc pcpal.c funs1.c
```

FUNCIONES ESTÁTICAS

Una función estática es aquella que sólo es reconocible dentro del archivo en que ha sido definida, pero no fuera de este. Sólo tiene sentido hablar de funciones estáticas cuando se trabaja con programas multiarchivo.

En el programa multiarchivo mostrado abajo, la función `cuadrado` es estática. El modificador `static` al inicio de su cabecera así lo indica. Esta función sólo será visible dentro del archivo en que fue definida, pero no desde otros. Puede ser invocada desde `funs1.c`, p.ej. desde la función `cubo`, pero no desde `main` y, en general, desde ninguna función declarada fuera de `funs1.c`.

<pre>#include <stdio.h> int cubo(int); int main(void) { int b = cubo(5); printf("b=%d\n", b); //125 }</pre>	<pre>static int cuadrado(int x) { return x*x; } int cubo(int x) { return x*cuadrado(x); }</pre>
<code>pcpal.c</code>	<code>funs1.c</code>

Incluso si incluyéramos el prototipo `int cuadrado(int)` en el archivo `pcpal.c` obtendríamos un error en tiempo de enlazamiento (*linking time*). El enlazador se quejaría al no poder encontrar la definición de `cuadrado` por ninguna parte. Tanta es la invisibilidad de `cuadrado` fuera de `funs1.c` que se podría incluso definir otra función `int cuadrado(int)` en `pcpal.c`. Esta no generaría ningún conflicto con su homónimo en `funs1.c` a causa de la naturaleza estática de esta última.

LA FUNCIÓN PRINCIPAL, `main`

Ya hemos hablado de funciones en general. Ahora nos falta dedicarle unas líneas a una función particular, la función `main`. Su presencia es obligatoria en todo programa. Esta es la primera función que se ejecuta cuando invocamos nuestros programas desde la línea de comandos.

Solamente existen dos cabeceras correctas para definir la función principal. Estas son:

```
int main(void)
{
    int argc, const char *argv[])
```

A diferencia de otras funciones que reciben y devuelven datos a la función llamadora, la función `main` puede recibir datos desde y retornar valores hacia el proceso que la invoca (p.ej., al shell bash, desde donde solemos ejecutar nuestros programas).

El valor de retorno de `main` puede ser cualquier entero entre 0 y 255. El cero normalmente se usa para indicar que el programa ha concluido con éxito. No hay necesidad de colocar explícitamente la sentencia `return 0` al final de `main`. Por defecto `main` retorna 0 al proceso llamador. En caso de invocar nuestros programas desde el shell bash, uno podría ver el valor retornado por los mismos ejecutando `$?` en la línea de comandos.

Es posible pasarle datos a un programa al momento de su ejecución. Por ejemplo, uno podría invocar un programa `miprog.out` desde el shell bash de la siguiente manera:

```
./miprog.out 3.14 "Hola amigo" -61
```

En este caso, todos los datos mostrados pueden ser capturados y utilizados en el código fuente del programa a través de los parámetros de `main`. Para ello la cabecera de `main` debe ser definida como:

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

`argc` y `argv` son asignados durante la invocación del programa. El primer parámetro, `argc`, tomará como valor el número de argumentos usados en la invocación; el segundo, `argv`, almacenará los valores de dichos argumentos. En el caso de la invocación mostrada arriba, el texto `./miprog.out` se guardará en `argv[0]`; "3.14", en `argv[1]`; "Hola amigo", en `argv[2]`; y "-61" en `argv[3]`. Dado que los datos pasados a un programa se almacenan como cadenas en `argv`, muchas veces se necesita hacer conversiones de tipo antes de utilizarlos.

Terminemos nuestros comentarios viendo el siguiente programa y las salidas que produce:

```

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

int main(int argc, const char *argv[])
{
    int suma=0;
    for(int i=1; i<argc; i++)
        suma += atoi(argv[i]);

    printf("La suma de sus datos es %d", suma);
}

```

```

User@DESKTOP-ETMGBI ~
$ ./miprog 5 8 10
La suma de sus datos es 23
User@DESKTOP-ETMGBI ~
$ ./miprog 15 -8 34 5
La suma de sus datos es 46
User@DESKTOP-ETMGBI ~
$ ./miprog 13 5.4 3.2
La suma de sus datos es 21
User@DESKTOP-ETMGBI ~
$ ./miprog -10 -20 -30 -40 -50
La suma de sus datos es -150

```

El programa calcula e imprime la suma de todos los argumentos usados durante su ejecución. Uno podría pasar cualquier número de datos a un programa, tal como se observa en el gráfico de la derecha. Para sumar estos datos el programa acumula sus valores en la variable `suma`. Note que el bucle ignora el elemento `argv[0]` a la hora de actualizar `suma`. `argv[0]` siempre contiene el nombre del archivo ejecutable (`miprog`, en nuestro caso); los datos de usuario empiezan a partir de `argv[1]`.

La función `atoi`, con prototipo en `stdlib.h`, transforma una cadena, p.ej. "235" o "13.9", en un número entero, p.ej. 235 o 13. Existen otras funciones de conversión similares, p.ej. `atof`, `atol`, `atoll`, etc., en el mismo archivo `stdlib.h`.

RECUSIVIDAD

En las ciencias de la computación existen dos acepciones para la palabra recursividad. Esta podría referirse a estructuras de datos recursivas o a funciones recursivas.

Una estructura de datos es recursiva cuando al menos uno de sus atributos es un puntero a una estructura del mismo tipo. Las estructuras de datos recursivas son los bloques fundamentales que nos permiten construir estructuras dinámicas enlazadas, como pilas, colas y una amplia variedad de listas enlazadas y árboles.

Por otro lado, las funciones recursivas son aquellas que se invocan a sí mismas. Este será el tema que estudiaremos en lo que resta del presente capítulo.

DIVIDE Y VENCERÁS

«Divide et impera» era una máxima comúnmente utilizada en la arena política en épocas del Imperio Romano. Con ello quería decirse que para mantener el poder político se tenía que dividir a la oposición de modo que sus líderes quedasen separados y con serias dificultades para reunir sus fuerzas contestatarias.

En el campo de la informática «Divide y Vencerás» es una estrategia para resolver problemas computacionales. Esta consiste en dividir un problema en subproblemas más pequeños, de modo que las soluciones de estos últimos puedan luego integrarse para construir la solución del problema original, mucho más complejo.

Nada impresionante hasta aquí. El lector podría estar pensando que la estrategia de dividir un problema en subproblemas es algo natural y común y, por lo tanto, no merecía un nombre propio tan histórico y rimbombante. En eso estoy de acuerdo con el lector. Pero aún hay algo que falta decir. La estrategia «Divide y Vencerás» adquiere su mayor potencial cuando el problema a atacar puede ser subdividido en otros subproblemas más pequeños y del mismo tipo que el problema original. En tales situaciones, la resolución de un problema complejo puede convertirse en una secuencia de simplificaciones que, tarde o temprano, terminará con uno o pocos problemas triviales, cuyas soluciones son obvias.

Como ejemplo analicemos el problema de calcular el máximo común divisor (mcd) de 120 y 90. Por una propiedad matemática conocida como regla de Euclides se sabe que el mcd de dos números tiene el mismo valor que el mcd del menor de ellos y su diferencia. Siendo así, puedo utilizar la estrategia Divide y Vencerás para calcular el mcd de 90 y 30 en lugar de calcular el mcd de 120 y 90. Al hacer esto no he resuelto el problema original. Simplemente lo he sustituido por otro más sencillo, que involucra números más pequeños, pero que es equivalente al original. Así, el problema original puede pasar por sucesivas simplificaciones, tal como se muestra a continuación:

$$\text{mcd}(120, 90) \rightarrow \text{mcd}(90, 30) \rightarrow \text{mcd}(30, 60) \rightarrow \text{mcd}(30, 30)$$

Al final resulta que el problema de calcular el mcd de 120 y 90 era equivalente al de calcular el mcd de 30 y 30. Este último problema es trivial y, por ello, ya no será necesario sustituirlo por otro, sino que será directamente resuelto: El mcd de dos números iguales es el mismo número, lo que implica que el mcd de 120 y 90 es 30.

Existen muchos problemas que pueden ser reemplazados por otros más pequeños del mismo tipo. Por ejemplo, calcular `fact(N)`, el factorial de un número, es lo mismo que calcular `N*fact(N-1)`; determinar el N -ésimo término de la serie de Fibonacci, `fibo(N)`, es lo mismo que determinar sus dos términos anteriores y sumarlos, `fibo(N-1) + fibo(N-2)`; buscar un número N dentro de una lista ordenada es lo mismo que buscarlo en la primera mitad de la lista (si N es menor que el elemento central de la lista), o buscarlo sólo en la segunda mitad (si N es mayor que el elemento central). En todos estos problemas se puede reemplazar el problema original sucesivamente, una y otra vez, hasta llegar eventualmente a problemas triviales. Para los tres ejemplos que acabo de mencionar, estos serían: `fact(1)=1`, `fibo(1)=1` y `fibo(2)=1`, y buscar N en una sublista de 1 elemento.

FUNCIONES RECURSIVAS

Una función recursiva es aquella que se invoca a sí misma. Las funciones recursivas son una manera natural y directa de implementar la estrategia Divide y Vencerás.

Generalizando el razonamiento anterior, se puede calcular el mcd, no sólo de 90 y 120, sino en general, de A y B , utilizando la siguiente función recursiva:

```
int mcd(int A, int B) {  
    if(A==B)  
        return A;  
    else if (A < B)  
        return mcd(A, B-A);  
    else  
        return mcd(B, A-B);  
}
```

Función recursiva `mcd`

En la definición de una función recursiva suele distinguirse dos partes:

- **El caso base** se refiere al problema trivial al que se llega luego de sucesivas simplificaciones. En este ejemplo, el caso base ocurre cuando los números cuyo máximo común divisor se quiere calcular son iguales, i.e. $A==B$. En dicho escenario, el dato a retornar sería este valor común.
- **El caso recursivo** se refiere a todas aquellas situaciones donde un problema puede ser sustituido por otro más sencillo. En nuestro caso, esto ocurre siempre que A y B son diferentes. En vez de calcular la solución de `mcd(A, B)` se prefiere calcular `mcd(A, B-A)`, cuando A es el menor de los números, o `mcd(B, A-B)`, cuando B sea el menor.

Como segundo ejemplo a discutir podríamos implementar una función que retorne el N -ésimo término de la serie de Fibonacci, definida como 1, 1, 2, 3, 5, 8, 13,... Dicha función sería así:

```

int fibo(int N) {
    if(N==1 || N==2)
        return 1;
    else
        return fibo(N-1) + fibo(N-2);
}

```

Función recursiva fibo

Aquí hay una pequeña diferencia con respecto al problema anterior. Antes, el problema de calcular el `mcd` de `A` y `B` era reemplazado por otro cuya solución era exactamente igual a la del problema original. En el presente problema hay un ingrediente nuevo. Encontrar las soluciones de los subproblemas `fibo(N-1)` y `fibo(N-2)` no significa que el problema `fibo(N)` ya ha sido resuelto. Estas soluciones aún deben ser integradas. En este caso, deben sumarse, `fibo(N-1)+fibo(N-2)`. Recién allí se lograría resolver `fibo(N)`.

La sustitución de un problema grande, `fibo(N)`, por otros más pequeños, `fibo(N-1)` y `fibo(N-2)`, eventualmente terminará con los problemas triviales `fibo(1)` y `fibo(2)`, que consisten en retornar el primer y segundo elemento de la serie de Fibonacci que, por definición, son 1 y 1.

Incluso algo tan ordinario como imprimir los números de 1 a `N` puede implementarse recursivamente. El siguiente ejemplo prueba lo que acaba de decirse:

```

int print(int N) { //imprime 1,...,N
    if(N > 0) {
        print(N-1); //imprime 1,...,N-1
        printf("%d", N);
    }
}

```

Función recursiva print

Según el código mostrado, imprimir los números de 1 hasta `N`, `print(N)`, es lo mismo que imprimir desde 1 hasta `N-1`, `print(N-1)`, y luego imprimir manualmente `N`, `printf("%d", N)`. En este ejemplo, el caso base está tácito (`N=0`). Cuando eventualmente se necesite imprimir los números de 1 hasta 0, `print(0)`, simplemente no se hará nada.

La elegancia de una implementación recursiva no va a la par con su eficiencia. Las implementaciones iterativas son, por lo general, mucho más eficientes en términos de consumo de memoria. Esto se debe a que cada vez que invocamos una función, sea o no recursiva, el sistema reserva cierto espacio de memoria para poder gestionar las variables locales, los parámetros, la dirección de retorno de la función y un amplio etcétera. Lamentablemente, la invocación de una función recursiva ocasiona una larga cadena de llamadas a sí misma, y cada una de estas llamadas termina ocupando espacio en memoria. En el peor de los casos, estas llamadas podrían llegar a requerir más espacio del que el sistema ha reservado para el programa. Esto causaría el conocido problema de **desbordamiento de pila** (*stack overflow*).

Se sabe que cualquier cómputo que pueda ser realizado por una función recursiva también puede ser implementado de forma iterativa (utilizando bucles `for`, `while`) y viceversa. Esto podría llevar al lector a preguntarse: ¿para qué usar funciones recursivas entonces, si puedo hacer lo mismo utilizando bucles? Obviamente no es una necesidad. Sin embargo, hay problemas que pueden ser resueltos muy fácilmente cuando se enfocan bajo un pensamiento recursivo.

PROBLEMAS RESUELTOS

1. Una función que retorna varios valores

Se desea transformar una temperatura dada en grados Celsius a las escalas Farenheit, Kelvin y Rankine. ¿Cómo podría implementar una función que retorne los tres valores pedidos?

Solución:

Nos gustaría poder implementar una función `Transforma` que retorne tres datos en una misma sentencia `return`. Pero, lamentablemente, las funciones en C sólo pueden retornar un valor. Sin embargo, es posible subsanar esta limitación usando el paso por referencia.

Recordando las siguientes equivalencias:

$$\frac{^{\circ}C}{5} = \frac{^{\circ}F - 32}{9} = \frac{R - 491.67}{9} = \frac{K - 273.15}{5}$$

la función `Transforma` quedaría como sigue:

```
void Transforma(double C, double *pF, double *pK, double *pR) {
    *pF = 9*C/5 + 32;
    *pK = C + 273.15;
    *pR = 9*C/5 + 491.67;
}

int main(void) {
    double C = 23.2;
    double F, K, R;

    Transforma(C, &F, &K, &R);

    printf("Temp. en Celsius = %.2f\n", C);
    printf("Temp. en Farenheit = %.2f\n", F);
    printf("Temp. en Kelvin = %.2f\n", K);
    printf("Temp. en Rankine = %.2f\n", R);
}
```

`Transforma` posee cuatro parámetros: Por el primero se pasará la temperatura en Celsius que se desea transformar; los otros tres se usarán para introducir tres variables vacías, `F`, `K` y `R`, para que sean asignadas dentro de `Transforma`. Para ello se necesita que `F`, `K` y `R` sean pasadas por por referencia, lo cual se logra al invocar `Transforma(C, &F, &K, &R)` desde el `main`.

Las tres direcciones pasadas a `Transforma` son almacenadas en los parámetros `pF`, `pK` y `pR`, respectivamente. Al desreferenciar estos parámetros, `*pF`, `*pK` y `*pR`, es posible modificar el valor de las variables `F`, `K` y `R` definidas fuera de `Transforma`, en el `main`. Gracias al paso por referencia, las variables `F`, `K` y `R` retienen sus valores incluso después de que `Transforma` termina su ejecución y retorna al `main`.

Los tres últimos parámetros de `Transforma` se denominan **parámetros de salida**. Se usan para permitir que `Transforma` pueda «sacar» datos hacia afuera, hacia la función llamadora. En contraposición, su primer parámetro sería un **parámetro de entrada**, porque permite a la función llamadora «introducir» datos dentro de `Transforma`.

2. Evaluando polinomios

Sea el polinomio:

$$P(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \dots + a_{N-1} x^{N-1} + a_N x^N$$

Implemente una función que reciba los coeficientes $\{a_0, a_1, a_2, \dots, a_{n-1}, a_n\}$ y un número x , para luego retornar el valor del polinomio evaluado en x . Para que la evaluación sea eficiente se le pide utilizar el esquema de Horner, que se muestra abajo:

$$P(x) = a_0 + x (a_1 + \dots + x (a_{N-2} + x (a_{N-1} + a_N x)) \dots)$$

Solución:

La evaluación de un polinomio por el método de Horner nos evita tener que calcular cada potencia x^m independientemente, con lo que se ahorra una buena cantidad de multiplicaciones.

Para implementar este esquema nos conviene construir el polinomio desde adentro hacia afuera, siguiendo la siguiente secuencia:

$$\begin{aligned} P &= 0 \\ P &= a_N x \\ P &= x (a_{N-1} + a_N x) \\ P &= x (a_{N-2} + x (a_{N-1} + a_N x)) \\ &\vdots \end{aligned}$$

En la secuencia mostrada, la variable P terminará finalmente con el valor $P(x)$ buscado. La secuencia se implementará con un bucle. Inicialmente P iniciará en cero y, en cada iteración, su valor será multiplicado por x luego de ser aumentado en a_i ($i=N, N-1, \dots, 2, 1$). El término independiente a_0 es el único que no se multiplica por x .

La implementación pedida se muestra a continuación:

```
double PHorner(double coef[], int N, double x) {
    double P = 0;
    for(int i=N-1; i>0; i--)
        P = (P + coef[i])*x;
    return P + coef[0];
}
```

`PHorner` recibe tres parámetros: `coef` es el arreglo que contiene los coeficientes del polinomio P ; `N` es el tamaño del arreglo; y `x` es el valor donde se evaluará el polinomio.

La sentencia `P = (P + coef[i])*x` es una generalización de las cuatro actualizaciones mostradas arriba. Note que como primero se necesita evaluar los coeficientes de los últimos términos, el bucle recorre desde `i=N-1` hasta `i=1`, para así evaluar primero `coef[N-1]`, luego `coef[N-2]` y así sucesivamente.

Como el coeficiente a_0 nunca se multiplica por x , debe ser añadido manualmente, luego de haber calculado $x (a_1 + \dots + x (a_{N-2} + x (a_{N-1} + a_N x)) \dots)$, o sea, después de que el bucle ya haya concluido.

3. Actualizar puntero dentro de una función

Definir una función que reciba un puntero genérico (`void*`) y un entero `N`. La función debe incrementar el valor del puntero en `N` bytes. El puntero alterado debe preservarse luego que el flujo del programa retorne a la función llamadora.

Solución:

Este es un problema de paso por referencia. Así como en un ejercicio anterior se tuvo que pasar `&F`, `&K` y `&R` para que `F`, `K` y `R` puedan ser actualizadas dentro de una función, ahora, para lograr que un puntero `ptr` sea modificado dentro de una función se deberá pasar su dirección, `&ptr`, un puntero a puntero.

En la función `Actualiza` mostrada abajo, el puntero `pptr` es la dirección de un puntero cuyo valor deseamos modificar. La modificación se realiza con la instrucción `*pptr = *pptr + N`.

Dentro del `main`, los punteros `p1` y `p2` son inicializados con las direcciones de `x` y `y`, respectivamente. La primera llamada a `Actualiza` hace que `p1` se incremente en 10 bytes; la segunda llamada hace lo propio con `p2`, tal como se observa de las salidas anotadas en el código.

```
void Actualiza(void **pptr, int N)
{
    *pptr = *pptr + N;
}

int main(void) {
    int x;
    float y;

    void *p1 = &x;
    Actualiza(&p1, 10);
    printf("%p %p\n", &x, p1); // imprime 0xfffffc1c 0xfffffc26

    void *p2 = &y;
    Actualiza(&p2, 10);
    printf("%p %p\n", &y, p2); // imprime 0xfffffc18 0xfffffc22
}
```

El código de `Actualiza` nos permite deducir que aunque los punteros genéricos (`void*`) no pueden ser desreferenciados, los punteros a puntero genérico (`void**`) sí. Esto podría ser confuso, pero tiene mucho sentido. Al desreferenciar un `void**` se obtiene un `void*`, cuyo tamaño será siempre conocido, 8 bytes.

Finalmente, para ganar un poco más de profundidad, discutamos qué habría pasado si se declaraba `p1` como `int*` y no como `void*`. En ese caso la llamada `Actualiza(&p1, 10)` habría provocado un error de tipos incompatibles. Esta función espera recibir la dirección de un puntero genérico, `void**`; pero en el hipotético caso que estamos discutiendo estaría recibiendo la dirección de un puntero a entero, `**int`, pues eso sería `&p1`. Aunque un puntero de tipo `int*` puede ser asignado sin problemas a uno de tipo `void*`, esto no implica que un puntero de tipo `int**` pueda ser asignado a un `void**`.

4. Palabras, palabras, palabras

Implemente una función que reciba una cadena y un separador, y retorne los fragmentos de la cadena que se encuentran entre separadores. Por ejemplo, si pasamos la cadena "uno,dos,tres" y usamos la coma como separador, la función debe retornar el arreglo {"uno", "dos", "tres"}.

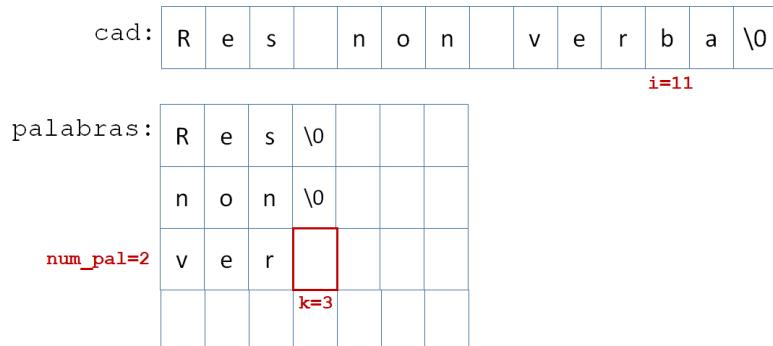
Solución:

Implementaremos la funcionalidad pedida con la función `dividir`, cuya cabecera será:

```
int dividir(char cad[], char sep, char palabras[][][20])
```

El parámetro `cad` es la cadena que se va a fragmentar usando el carácter `sep` como separador. Las palabras contenidas en `cad` se almacenarán como filas de la matriz `palabras`. La función retornará el número de palabras halladas en `cad`. Se asume que cada palabra no tiene más de 20 caracteres.

A continuación comentaré la estrategia a seguir ayudándome del siguiente diagrama:



Deberán visitarse todos los caracteres de `cad` hasta llegar al carácter nulo '\0'. Cada carácter visitado debe ser añadido a la casilla `palabras[num_pal][k]`. La clave para implementar la función pedida consiste en saber actualizar `num_pal` y `k`.

El valor de `num_pal` representa el número de palabras ya encontradas; su valor inicial es cero pero se incrementa cada vez que se encuentra un separador en `cad`. En el diagrama se ha usado el espacio en blanco como separador.

El valor de `k` representa el número de caracteres de la palabra que está siendo desvelada; inicia en cero, se incrementa de uno en uno, y vuelve a hacerse cero cada vez que se encuentra un separador.

El código de la función quedaría como sigue:

```

int dividir(char cad[], char sep, char palabras[][][20]) {
    int num_pal=0;
    int k = 0;

    for(int i=0; cad[i] != '\0'; i++) {
        if (cad[i]==sep) {
            palabras[num_pal][k]='\0';
            k = 0;
            num_pal++;
        } else
            palabras[num_pal][k++] = cad[i];
    }
    palabras[num_pal++][k] = '\0';
    return num_pal;
}

```

Dentro del bucle `for`, cada carácter `cad[i]` será visitado uno por uno hasta llegar al carácter nulo.

Normalmente la función `dividir` copia el carácter visitado en una celda de la matriz `palabras` haciendo `palabras[num_pal][k++]=cad[i]`. El operador `++` mueve el cursor (digámoslo así) a la derecha, preparándose para escribir el siguiente carácter de la misma palabra. El caso especial ocurre cada vez que se encuentra el carácter separador, i.e. `cad[i]==sep`. En dicho momento, se debe cerrar la palabra anterior añadiendo el carácter nulo al final de su contenido, `palabras[num_pal][k]='\0'`. Y, adicionalmente, se debe incrementar `num_pal` y reiniciar `k` porque justo en ese momento se empezará a escribir el primer carácter (`k=0`) de una nueva palabra.

Pero no todas las palabras acaban con el separador `sep`. La última palabra en `cad` termina con el carácter nulo. Sin embargo, también es necesario cerrar esta última palabra y contabilizarla. Ambas tareas se realizan con `palabras[num_pal++][k]='\0'`, fuera del bucle `for`

El retorno de `num_pal` a la función llamadora sirve para que ésta conozca cuantas palabras fueron almacenadas en la matriz pasada como parámetro a la función `dividir`.

Abajo mostramos cómo se podría invocar a la función `dividir`.

```

int main(void) {
    char pals[10][20];
    char msg[]="Res non verba";
    int N = dividir(msg, ' ', pals);
    for(int i=0; i<N; i++)
        printf("%s\n", pals[i]);
}

```

La matriz `pals` debería tener más filas que el número de palabras que se espera encontrar en `cad`.

Las `N` palabras halladas en `cad` son impresas dentro del bucle, al final del `main`. Cada palabra encontrada (i.e. cada fila de la matriz `pals`) puede imprimirse con el especificador `%s`. Si no se hubieran colocado los caracteres nulos `'\0'`, las impresiones de `pals[i]` podrían haber arrojado caracteres extraños al final de cada palabra.

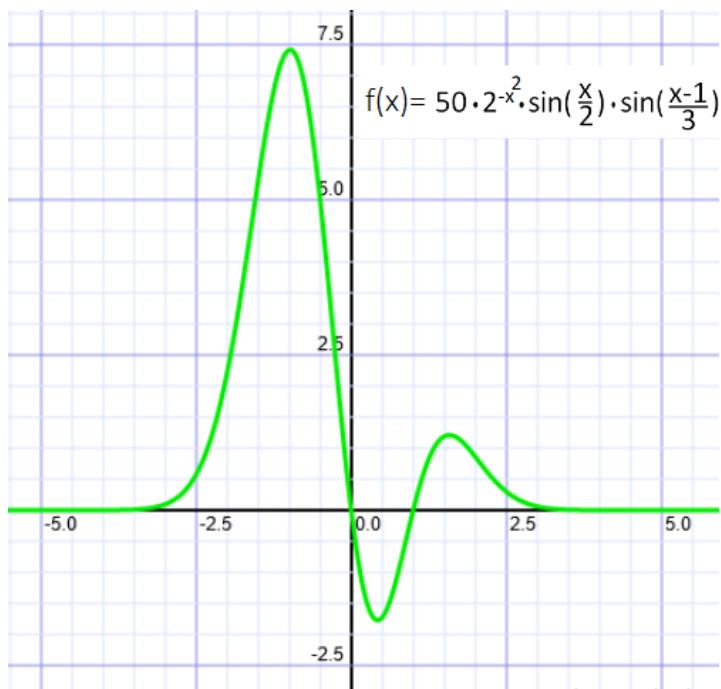
5. Hallando los máximos de una función

El método de optimización de trepacolinas estocástico (Stochastic Hill Climbing) es una estrategia heurística que intenta encontrar los máximos locales de una función continua. Esta estrategia consta de los siguientes pasos:

- a. Iniciar la búsqueda en cualquier punto $x=x_0$ del dominio de f
- b. Aplicar una pequeña perturbación a x , obteniendo x'
- c. Si $f(x') > f(x)$, actualice $x=x'$. De lo contrario, ejecute (b).

Una perturbación consiste en añadirle un pequeño valor aleatorio (positivo o negativo) a x . Para este ejercicio haremos que el algoritmo concluya cuando en 10 perturbaciones consecutivas no se pueda encontrar un mejor valor para $f(x)$. En este momento se asumirá que al ya no poder encontrar mejores valores en el vecindario de x , este sería un máximo local.

Implemente un programa que encuentre el máximo de la siguiente función con el método descrito:



Solución:

El método trepacolinas enfoca un problema de optimización como uno de búsqueda. Podríamos imaginar la búsqueda del máximo global como una caminata en la que uno se mueve sobre el eje X a partir de un punto arbitrario $x=x_0$. Estando sobre el punto x , uno podría dar pequeños pasos de longitud variable, hacia la derecha o izquierda, pero sólo se asentará la pisada en $x+\text{eps}$ si se cumple $f(x+\text{eps}) > f(x)$. El valor eps es generado aleatoriamente. Esto le concede el adjetivo de estocástico al método descrito.

La función a optimizar puede implementarse de la siguiente forma:

```
double f (double x)
{
    return 50*pow(2, -x*x)*sin(x/2)*sin((x-1)/3);
```

El método heurístico descrito puede implementarse así:

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

double f (double x) {
    return 50*pow(2, -x*x)*sin(x/2)*sin((x-1)/3);
}

int main(void) {
    double x=0;
    double eps;
    int mov_sin_mejor = 0;

    srand(time(NULL));

    do {
        eps = pow(-1, rand())*0.1*(double)rand()/RAND_MAX;
        if(f(x+eps) > f(x)) {
            x = x+eps;
            mov_sin_mejor = 0;
        } else
            mov_sin_mejor++;
    } while(mov_sin_mejor < 10);

    printf("\nx*=%f, f(x*)=%f", x, f(x));
}
```

Dentro del `main`, la búsqueda se inicia en el punto `x=0`. En cada iteración se añade `eps` a `x`. `eps` es un número aleatorio cuyo valor absoluto oscila entre 0 y 0.1. Se calcula con la expresión `0.1*(double)rand()/RAND_MAX`. El signo de `eps` puede ser positivo o negativo, según lo decida el azar. El signo se calcula con `pow(-1, rand())`.

Si proyectásemos nuestra caminata en el eje X sobre la curva `f`, esta siempre consistiría de una serie de pasos que conducen hacia arriba (de allí el nombre del método, trepapolinas). El valor `x` sólo es actualizado a `x+eps` si este nuevo valor permite ascender sobre la curva `f`, i.e. si $f(x+eps) > f(x)$.

La variable `mov_sin_mejor` permite contar las perturbaciones consecutivas fallidas que se realizan a partir de un determinado `x`, i.e. las veces que un distinto `eps` no conduce a $f(x+eps) > f(x)$. Cuando esta variable toma el valor de 10 se acepta sin pruebas que ya no hay mejores valores que `x` en su vecindario y, por ende, se termina la búsqueda, asumiendo que dicho `x` es una buena aproximación de algún óptimo local de la función `f`.

Luego de ejecutar 1000 veces el programa anterior desde $x=0$, el promedio obtenido para el óptimo encontrado x^* es -0.974841 , donde $f(x^*)=7.413869$. Esto está cerca al óptimo global del gráfico mostrado. Pero cuando se ejecuta el programa 1000 veces partiendo desde $x=3$, el óptimo promedio que se encuentra es $x^*=1.599223$ y $f(x^*)=1.208448$, cerca del óptimo local.

Una característica de los métodos de trepacolinias (*hill climbing*) es que se estancan en óptimos locales, sólo alcanzan el óptimo global cuando parten de un buen punto de inicio x_0 . Este estancamiento suele ser resuelto por otros algoritmos heurísticos más sofisticados (p.ej. *Simulated Annealing*) que, bajo ciertas condiciones, aceptan caminar hacia peores soluciones con la esperanza de escapar de los óptimos locales hacia el óptimo global.

6. Sacando raíz cuadrada

Obtener la raíz de cuadrada de x equivale a encontrar un y tal que: $y^2=x$ o, en otras palabras, un y que satisface $y=x/y$. Aprovechando esta observación se puede implementar un algoritmo que nos aproxime a la raíz de x . Este algoritmo consta de los siguientes pasos:

- Asignar a y cualquier valor positivo, p.ej. una estimación intuitiva de la raíz buscada.
- Si y se approxima a x/y , ya se ha encontrado una solución aproximada. Fin del cálculo.
- Si y no se approxima a x/y tanto como se desea, haga $y = \frac{1}{2}(y + x/y)$ y vuelva al paso b.

Implemente el algoritmo que se acaba de describir.

Solución:

El algoritmo descrito puede implementarse recursivamente. El caso base sería cuando el valor estimado y está muy próximo a x/y , digamos que a menos de una millonésima de distancia. En este caso ya podríamos considerar que y es una buena aproximación de la raíz cuadrada de x y retornar dicho valor. El caso recursivo ocurre cuando y aún está lejos de x/y . Aquí todavía se debe tantear un nuevo valor para y . Este sería el punto medio entre los valores antes comparados, es decir, la semisuma de y e x/y .

El siguiente código implementa la recursividad descrita:

```
double RaizPorTanteo(double x, double y) {
    if(fabs(y-x/y)<0.000001)
        return y;
    else {
        y = 0.5*(y + x/y);
        return RaizPorTanteo(x, y);
    }
}
```

La función `fabs`, con prototipo en `math.h`, retorna el valor absoluto de un número decimal. Como ya se dijo, si x e y están a una distancia menor de 0.000001 se considerarán iguales. El nivel de tolerancia también pudo haber sido pasado como parámetro.

El siguiente código muestra cómo calcular las raíces cuadradas de los números de 1 a 10, iniciando el tanteo siempre a partir de 1 (segundo parámetro de `RaizPorTanteo`). En todos los casos se observa una coincidencia de 6 decimales con la raíz calculada mediante `sqrt` (en `math.h`).

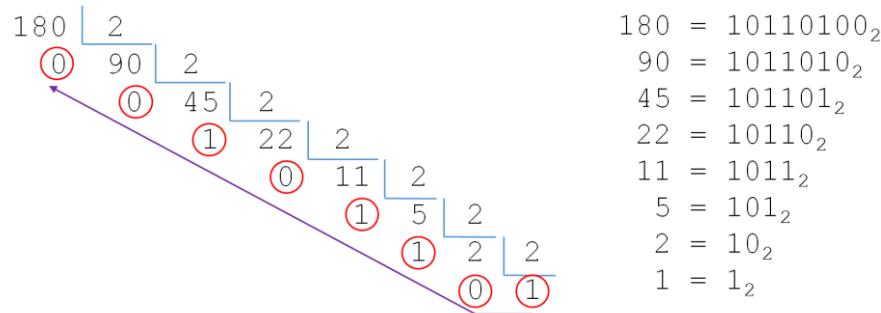
```
int main(void)
{
    for(int i=1; i<=10; i++)
        printf("raiz(%d)= %f %f\n", i, RaizPorTanteo(i, 1), sqrt(i));
}
```

7. Cambio de base recursivo

Implemente una función recursiva que imprima el número N en base b ($b < 10$).

Solución:

Concentrémonos en la transformación por divisiones sucesivas del número $N=180$ a la base $b=2$. En el gráfico de la derecha se observa fácilmente la recursión. La tarea de imprimir $N=180$ en binario puede ser vista como dos subtareas: (1) imprimir $N=90$ en binario, y (2) imprimir un cero a la derecha. Esto es así porque la secuencia de residuos que se obtiene al dividir sucesivamente 180 por 2 es la misma que la que se obtiene al dividir 90 sucesivamente por 2, excepto que la representación binaria de 180 tiene un bit adicional al final, el residuo de 180 entre 2; en nuestro caso, cero.



En general, imprimir N en base b es lo mismo que (1) imprimir el cociente de N/b en base b , para luego (2) imprimir un dígito extra, el residuo de la división N/b . El caso trivial de imprimir N en base b ocurre cuando N es menor que la base b . En dicho caso, se debe imprimir directamente N , sin tener que hacer ningún cálculo. La función `CambiaBase` se implementa abajo:

```
void CambiaBase(int N, int b) {
    if(N < b)
        printf("%d", N); // caso base
    else {
        CambiaBase(N/b, b);
        printf("%d", N % b);
    }
}
```

Esta función sólo puede transforma N a bases $b < 10$. Si desea trabajar con bases $b < 16$, tiene que cambiar el especificador de formato %d por %x en las dos sentencias printf.

8. Imprimir una cadena al revés

Implemente una función recursiva que imprima una cadena al revés.

Solución:

Imprimir una cadena al revés es lo mismo que: (1) imprimir su último carácter, para, seguidamente, (2) imprimir el resto de la cadena al revés. Esto se implementa así:

```
void invierte(char* cad, int N) {
    if(N > 0)
    {
        printf("%c", *(cad+N-1));
        invierte(cad, N-1);
    }
}
```

La cadena a invertir posee N caracteres; el primero se encuentra en la dirección cad.

La función empieza imprimiendo el último carácter de la cadena, el cual se encuentra a $N-1$ bytes de distancia del primero, o sea, en la dirección $cad+N-1$. Dicho carácter es $* (cad+N-1)$.

A continuación la función imprime (al revés) la subcadena restante. Como la función invierte es tal que imprime al revés los N caracteres a partir de cad, la subcadena restante (aquella sin el carácter final, ya impreso) se imprimiría con invierte(cad, N-1).

El caso trivial consiste invertir una cadena nula, $N=0$, para lo cual no se necesitaría hacer nada. Esa es la razón por la que no hay una especificación explícita de un caso base en el código. No se necesita escribir instrucciones para decirle al sistema que no haga nada. La función invierte puede invocarse de las siguientes maneras:

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

int main(void) {
    char txt1[20] = "William Shakespeare";
    invierte(txt1, strlen(txt1));

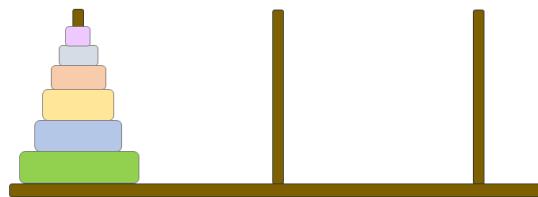
    char *txt2 = "amor a roma";
    invierte(txt2, strlen(txt2));
}
```

9. Torres de Hanoi

El juego de las torres de Hanoi involucra tres postes y N discos de distintos tamaños. Inicialmente todos los discos se encuentran apilados en uno de los postes, el poste izquierdo, de modo que los discos más pequeños se encuentran más arriba en la pila (ver Figura).

El objetivo del juego consiste en trasladar los N discos desde poste izquierdo hacia el poste derecho respetando dos reglas:

- Sólo se puede mover un disco a la vez, el de la parte superior de la pila.
- Nunca se debe colocar un disco sobre otro más pequeño.



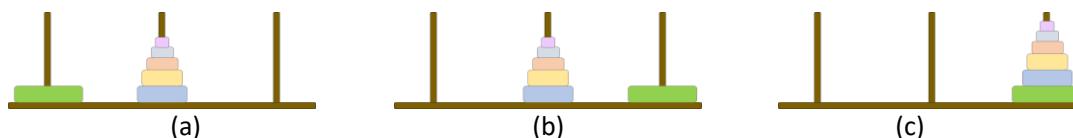
Una de las muchas historias asociadas a este juego asegura que son 64 discos de oro los que, desde el inicio de la Creación, vienen siendo movidos a razón de un disco por día por unos monjes fanáticos en algún arcano templo de la vaporosa Hanoi, capital de Vietnam. La misma leyenda afirma que el mundo se acabará apenas los ermitas concluyan su trabajo.

Implemente un programa que muestre la secuencia de movimientos que permitiría trasladar correctamente N discos desde poste izquierdo al poste derecho.

Solución:

Pensando en la salida del programa podemos representar cada poste con un número: 1 para el poste izquierdo, 2 para el central, y 3 para el de la derecha. Así, cada movimiento a realizar puede representarse haciendo referencia a estos números. Por ejemplo, la solución para $N=2$ discos consistiría de solo tres movimientos: Mover un disco al poste central (1→2), mover el otro disco al poste derecho (1→3), y, finalmente, mover el disco que habíamos dejado en el poste central hacia el poste derecho (2→3). Observe que esta notación no especifica qué disco debe moverse. No es necesario. Por la forma como están apilados los discos se sobreentiende que sólo puede moverse el disco que está en la parte superior de cada poste.

La manera más simple de resolver este problema es haciendo uso del pensamiento recursivo, según el cual, mover N discos del poste izquierdo al derecho equivale a: (a) mover $N-1$ discos hacia el poste central, (b) mover el disco restante hacia el poste derecho, y (c) mover $N-1$ discos del poste central al derecho, tal como se ilustra abajo:



No hay ninguna trampa en el procedimiento descrito. Sabemos que el movimiento de $N-1$ discos del poste inicial al central está prohibido; pero esto puede resolverse moviendo previamente $N-2$ discos

al poste derecho, el disco restante al poste central, y devolviendo los $N-2$ discos del poste derecho al centro, y así recursivamente.

Siguiendo la regla descrita, el traslado de una torre de N discos puede verse como el traslado sucesivo de varias subtorres cada vez más pequeñas y, eventualmente, de un único disco.

La estrategia descrita se implementa con el siguiente código:

```
void hanoi(int N, int pini, int pfin, int ptmp) {  
    if(N==1)  
        printf("%d-->%d\n", pini, pfin);  
    else {  
        hanoi(N-1, pini, ptmp, pfin);  
        hanoi( 1 , pini, pfin, ptmp);  
        hanoi(N-1, ptmp, pfin, pini);  
    }  
}
```

Una llamada a `hanoi(N, pini, pfin, ptmp)` imprimirá la secuencia de movimientos que se requiere para trasladar N discos desde el poste `pini` al poste `pfin` utilizando el poste `ptmp` como auxiliar. Los parámetros `pini`, `pfin` y `ptmp` sólo pueden tomar los valores 1, 2 y 3.

El problema es trivial cuando se trata de mover un único disco ($N=1$). En dicho caso se imprime directamente la solución, el movimiento de `pini` a `pfin`.

En otros casos, el problema de mover N discos de `pini` a `pfin` se resuelve en tres pasos: (1) mover $N-1$ discos de `pini` a `ptmp`, (2) mover el único que quedaba en `pini` a `pfin`, y (3) mover los $N-1$ discos que estaban en `ptmp` hacia `pini`.

La solución pedida se obtiene invocando `hanoi` de la siguiente manera:

```
int main(void){  
    int N;  
  
    printf("Ingrese número de discos:");  
    scanf("%d", &N);  
    printf("Puede mover %d discos de 1 a 3 haciendo:\n", N);  
    hanoi(N, 1, 3, 2);  
}
```

Es sabido que el número de movimientos necesarios para trasladar una torre de N discos de un poste a otro es 2^N-1 , tal como puede verificarse ejecutando el programa para distintos valores de N .

Finalmente, dejamos una pregunta al lector: si la leyenda vietnamita fuese cierta, ¿cuánto tiempo faltaría para el fin del mundo?

10. Búsqueda binaria

La búsqueda binaria es un algoritmo que permite buscar un número en una lista ordenada sin tener que visitar todos los elementos de la lista. Consiste en comparar el elemento central de la lista con el número buscado. Si el elemento central es menor, esto significa que el número buscado tendría que estar en la sublista derecha (junto con los números grandes), y hacia ese sector deberíamos redirigir la búsqueda. De lo contrario, deberíamos concentrarnos en la sublista izquierda. Eventualmente, conforme evaluamos sublistas cada vez más pequeñas, el número buscado será encontrado en el centro de alguna de estas. Implemente una función recursiva con el algoritmo descrito.

Solución:

Dado que la búsqueda sobre un arreglo inicial se reducirá a búsquedas sobre subarreglos más pequeños, es importante definir una función que nos permita buscar en cualquier porción de un arreglo arbitrario.

La función `busqueda`, mostrada abajo, buscará el elemento `elem` dentro del subarreglo cuyo primer y último elemento son: `array[izq]` y `array[der]`, respectivamente. Así, para empezar la búsqueda de `elem` sobre un arreglo `arr` se deberá invocar `busqueda(arr, 0, N-1, elem)`, donde `N` es el número de elementos de `arr`.

```
int busqueda(int array[], int izq, int der, int elem) {
    if(izq == der)
        return array[izq] == elem ? izq : -1;
    else {
        int med = (izq + der)/2;
        if(elem == array[med])
            return med;
        else if(elem < array[med])
            return busqueda(array, izq, med, elem);
        else
            return busqueda(array, med+1, der, elem);
    }
}
```

Eventualmente la búsqueda podría llegar a reducirse a un subarreglo de un único elemento (`izq==der`). En este caso trivial es fácil saber si el elemento buscado está o no en dicho subarreglo. Basta comparar `elem` con `array[izq]` (o `array[der]`, que es lo mismo). Si ambos son iguales se retorna `izq` (o `der`), la posición donde se encuentra `elem`; de lo contrario se retornaría `-1`, como indicativo de que `elem` no está en la lista.

Pero, en el caso general, la búsqueda sobre un arreglo termina redireccionándose hacia alguno de sus subarreglos, derecho o izquierdo. Para ello debe compararse el elemento a buscar, `elem`, con el elemento central del arreglo, aquel ubicado en el subíndice `med=(izq+der)/2`. De esta comparación podría ocurrir que ambos valores sean iguales, `array[med]==elem`. En dicho caso habríamos encontrado lo que estábamos buscando. La función retornará la posición de `elem`, i.e. `return med`. Pero si no somos tan afortunados, el elemento central, `array[med]`, no coincidirá con `elem`. Podría ocurrir que `elem<array[med]`, en cuyo caso la búsqueda debe dirigirse al subarreglo izquierdo, aquel cuyos elementos ocupan las posiciones desde `izq` hasta `med`. No tendría sentido examinar los elementos del subarreglo derecho, pues todos ellos son mayores que `array[med]` y, obviamente,

`elem` no podría estar allí. No olvide que el arreglo está ordenado (y, según hemos asumido, ascendenteamente).

También podría ocurrir lo contrario, `elem > array[med]`, en cuyo caso la búsqueda debe ser redirigida al subarreglo derecho, aquel cuyos índices van desde `med+1` hasta `der`.

11. Ordenación por fusión, mergesort

Una conocida estrategia recursiva que permite ordenar los elementos de un arreglo es la siguiente: Primero se ordena la mitad derecha arreglo, luego la mitad izquierda, y finalmente se fusionan ambos subarreglos ya ordenados. Se le pide ordenar ascendenteamente los elementos de un arreglo implementando la estrategia descrita.

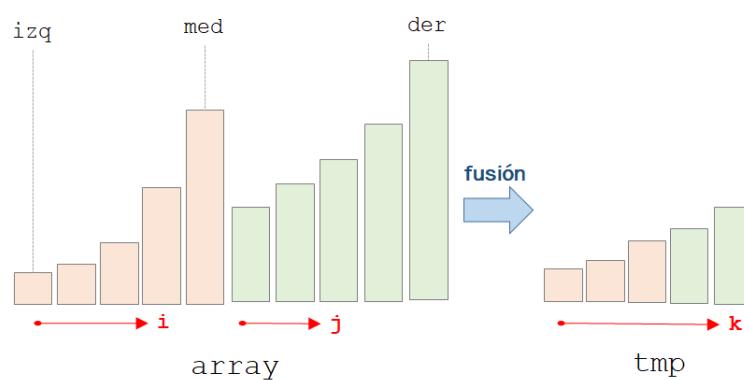
Solución:

Podemos definir una función `mergesort` para que ordene la porción de `array` cuyos subíndices van desde `izq` hasta `der`. El código se muestra abajo:

```
void mergesort(int array[], int izq, int der)
{
    if(izq < der) {
        int med = (izq + der)/2;
        mergesort(array, izq, med);
        mergesort(array, med+1, der);
        fusion(array, izq, der, med);
    }
}
```

Para ordenar el subarreglo de `array` que empieza en el subíndice `izq` y termina en `der` se debe encontrar el punto medio, `med`, entre `izq` y `der`, para luego (1) ordenar el subarreglo que va desde `izq` hasta `med`, (2) ordenar el subarreglo que va desde `med+1` hasta `der` y, finalmente, (3) fusionar ambos subarreglos ya ordenados.

La fusión de subarreglos ordenados es el paso más difícil. Lo explicaré con el siguiente gráfico:



Para mezclar dos subarreglos ordenados se debe comparar el i -ésimo elemento del subarreglo izquierdo con el j -ésimo elemento del subarreglo derecho. El menor de ambos debe agregarse a un arreglo temporal `tmp` que, al final del procedimiento, terminará siendo el arreglo ordenado que buscamos.

La variable `i` inicia con el valor `izq` y se incrementa cada vez que el elemento `arr[i]` es transferido a `tmp`; la variable `j` inicia con el valor `med+1` y se incrementa cada vez que `arr[j]` pasa a `tmp`; la variable `k` inicia en cero y aumenta cada vez que se inserta un elemento en `tmp`.

Eventualmente ocurrirá uno dos eventos: O la variable `i` sobrepasará el punto medio `med`, o el valor de `j` sobrepasará `der`. Lo primero significaría que todos los elementos del subarreglo izquierdo ya han sido transferidos a `tmp` y, por lo tanto, los elementos restantes del subarreglo derecho, que en ese momento estarían a la derecha del subíndice `j`, deberán ser pasados directamente a `tmp`, sin necesidad de comparaciones adicionales, pues ya no hay más elementos contra los cuales compararlos. Algo análogo debe ocurrir cuando `j` sobrepase el valor `der`.

El código de la función `fusion` se muestra abajo:

```
void fusion(int array[], int izq, int der, int med) {
    int tmp[der-izq+1];
    int i=izq, j=med+1;
    int k=0;

    while(i<=med && j<=der)
        if(array[i] < array[j])
            tmp[k++] = array[i++];
        else
            tmp[k++] = array[j++];

        if(i>med)
            while(j<=der) tmp[k++] = array[j++];
        else
            while(i<=med) tmp[k++] = array[i++];

    for(int i=izq; i<=der; i++) array[i] = tmp[i-izq];
}
```

El arreglo `tmp` se define como un VLA de `der-izq+1` elementos, cuyos valores son asignados dentro de un bucle `while`. El elemento `tmp[k]` se asigna con el menor valor entre `arr[i]` y `arr[j]`. El bucle termina cuando uno de los índices sobrepasa su límite; es decir, cuando deja de cumplirse `i<=med && j<=der`.

Como ya se dijo, si todos los elementos del subarreglo izquierdo ya han sido transferidos al arreglo temporal, `if(i>med)`, los elementos del subarreglo derecho que aún no han sido comparados deben pasar directamente a `tmp`. Esto se implementa en el bucle `while(j<=der)`. En caso contrario, si fueran los elementos del subarreglo derecho los primeros en copiarse a `tmp`, la parte del subarreglo izquierdo que nunca llegó a ser comparada debería pasar directamente a `tmp`, lo que se implementa dentro del bucle `while(i<=med)`.

En la última línea el arreglo `tmp` se copia en el arreglo original `array`.

La función `mergesort` puede invocarse con el siguiente código:

```

int main(void) {
    int arr[10] = {1, 10, 30, 5, 12, 34, 78, 21, 18, 11};
    mergesort(arr, 0, 9);
}

```

El lector puede añadir el código necesario para verificar que luego de la llamada a `mergesort` el arreglo `arr` queda ordenado.

12. Los caballos eran fuertes, los caballos eran ágiles

El problema del recorrido del caballo tiene más de mil años. Consiste en determinar si es posible que un caballo de ajedrez visite todas las casillas de un tablero de `FILS` filas y `COLS` columnas en `FILSxCOLS` pasos; es decir, sin pisar la misma casilla dos veces. Implemente un programa que resuelva este antiguo acertijo.

Solución:

Desde una casilla cualquiera de un tablero un caballo podría usar su movimiento en forma de L para llegar como máximo a otras ocho casillas, tal como se muestra en el siguiente gráfico:



Podríamos representar un tablero de `FILSxCOLS` casillas como una matriz de las mismas dimensiones. En cuanto al recorrido del caballo, se podría anotar la casilla (i, j) de la matriz con el valor n para indicar que el caballo ha visitado dicha casilla en su n -ésimo salto. Por ejemplo, las matrices que se muestran abajo representan dos posibles recorridos sobre un tablero de 3×4 . Siga la secuencia 1, 2, 3,... para ver cada recorrido.

$\begin{matrix} 1 & 4 & 7 & 10 \\ 8 & 11 & 2 & 5 \\ 3 & 6 & 9 & 12 \end{matrix}$	$\begin{matrix} 1 & 4 & 7 & 10 \\ 12 & 9 & 2 & 5 \\ 3 & 6 & 11 & 8 \end{matrix}$
Recorrido 1	Recorrido 2

La función `SaltaSobre`, mostrada abajo, imprime todos los posibles recorridos de caballo para ciertos parámetros dados.

```

void SaltaSobre(Tablero t, int i, int j, int n) {
    if(t.elems[i][j] == 0) {
        t.elems[i][j] = n;
        if(n == FILS*COLS)
            imprimeTablero(t.elems);
        else {
            if(dentroTablero(i-1, j-2)) SaltaSobre(t, i-1, j-2, n+1);
            if(dentroTablero(i-2, j-1)) SaltaSobre(t, i-2, j-1, n+1);
            if(dentroTablero(i-2, j+1)) SaltaSobre(t, i-2, j+1, n+1);
            if(dentroTablero(i-1, j+2)) SaltaSobre(t, i-1, j+2, n+1);
            if(dentroTablero(i+1, j+2)) SaltaSobre(t, i+1, j+2, n+1);
            if(dentroTablero(i+2, j+1)) SaltaSobre(t, i+2, j+1, n+1);
            if(dentroTablero(i+2, j-1)) SaltaSobre(t, i+2, j-1, n+1);
            if(dentroTablero(i+1, j-2)) SaltaSobre(t, i+1, j-2, n+1);
        }
    }
}

```

`SaltaSobre` evalúa si el caballo puede realizar su n -ésimo salto sobre la casilla (i, j) de un tablero t . El caballo realiza su salto, i.e. $t.elems[i][j]==n$, si dicha casilla aún no ha sido visitada; es decir si $t.elems[i][j]==0$. Inicialmente, el tablero t es una matriz nula.

En el mejor de los casos, el salto realizado podría haber sido el último, $n==FILS\times COLS$. De ser así, ya se tendría un recorrido completo y sólo faltaría imprimir el tablero t . Así veríamos un nuevo recorrido de caballo, que estaría codificado como las matrices mostradas arriba.

En el caso general, luego de realizar el n -ésimo salto, se debe evaluar dónde podría el caballo estampar sus cascós por $n+1$ -ésima vez. De las ocho posibles casillas a las que podría saltar, la función `dentroTablero` descarta aquellas que no son factibles, por quedar fuera del tablero.

```

int dentroTablero(int fil, int col) {
    return col>=0 && col<COLS && fil>=0 && fil<FILS;
}

```

En la primera invocación a `SaltaSobre` debe indicarse la posición de partida del caballo. Por ejemplo, según el código mostrado abajo el caballo partirá desde $(0, 0)$ en su $n=1$ -er salto. En ese momento inicial el tablero t está lleno de ceros, $t=\{0\}$.

```

int main(void) {
    Tablero t = {0};
    SaltaSobre(t, 0, 0, 1);
}

```

Las dimensiones del tablero están definidas en constantes simbólicas, fuera de todas las funciones.

```

#define FILS 3
#define COLS 4

```

Tal como se deduce del código mostrado, el tablero no está representado como matriz sino como estructura que contiene una matriz. Este artificio de pasar una matriz dentro de una estructura nos permite pasar la matriz por valor, que es lo que conviene en este problema. Si hubiésemos utilizado el paso por referencia, todas las llamadas recursivas habrían escrito sobre una misma matriz y habría sido difícil borrar las secuencias de saltos que al final terminaron conduciendo a recorridos fallidos.

La estructura Tablero fue definida así:

```
typedef struct tablero {
    int elems[FILS][COLS];
} Tablero;
```

El atributo `elems` ocupa su propio espacio de memoria en cada llamada a `SaltaSobre`.

13. Manipulación de bits

Implementar una función que retorne el N -ésimo bit del byte que ocupa la dirección `ptr`. Los valores `N` y `ptr` son parámetros de la función.

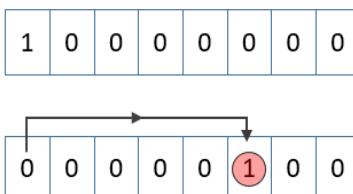
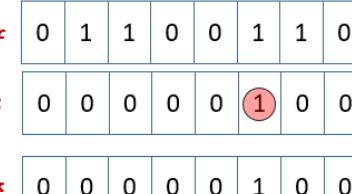
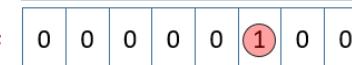
Solución:

A diferencia de otros lenguajes, C nos permite manipular los bits que conforman la representación interna de una variable. Para esto ofrece los siguientes operadores:

Operador	Nombre	Descripción
<code>A B</code>	OR	La operación OR es aplicada entre cada bit de A y su correspondiente bit en B. $b_1 b_2$ es 1 si alguno de los bits involucrados lo es, y 0 en otro caso.
<code>A & B</code>	AND	La operación AND es aplicada entre cada bit de A y su correspondiente bit en B. $b_1 \& b_2$ es 1 si ambos bits involucrados son 1's, y 0 en otro caso.
<code>A ^ B</code>	OR exclusivo	La operación EXCLUSIVE-OR se aplica entre cada bit de A y su correspondiente bit en B. $b_1 ^ b_2$ es 1 si sólo uno de los bits es 1's, y 0 en otro caso.
<code>A >> n</code>	desplazamiento a la derecha	Cada bit de A es desplazado n espacios a la derecha. Las posiciones iniciales de los bits desplazados se llenan con ceros cuando $A > 0$.
<code>A << n</code>	desplazamiento a la izquierda	Cada bit de A es desplazado n espacios a la izquierda. Las posiciones iniciales de los bits desplazados se llenan con ceros.
<code>~A</code>	negación	Cada bit de A es invertido. Los 0's se convierten en 1's y viceversa.

Para la implementación pedida sólo necesitaremos usar dos de estos operadores: `&` y `>>`.

La estrategia que usaremos para extraer el N -ésimo bit de un byte consistirá en construir una secuencia de 8 bits, todos ceros, excepto por el N -ésimo bit, que será 1. Esta cadena de bits suele denominarse **máscara** y sirve para fijar la posición del bit a extraer. Así, por ejemplo, para extraer el tercer bit de un byte debemos construir la máscara 00100000; para extraer el quinto, la máscara 00001000, etc., tal como se muestra en la parte izquierda del siguiente gráfico:

mask=128  mask=128>>5 	valor  mask  valor & mask 
El operador de desplazamiento <code>>></code> desplaza cinco espacios a la derecha cada bit de la variable <code>mask</code>	El operador <code>&</code> aplica la operación AND entre cada bit de <code>valor</code> y su bit correspondiente en <code>mask</code>

Posteriormente debemos aplicar el AND-BINARIO (operador `&`) entre la máscara y el byte que queremos evaluar. Luego de esta operación sólo existen dos posibilidades: O todos los bits del resultado son ceros, o hay un bit 1 en la N -ésima posición.

El código pedido es como sigue:

```
char obtenerNbit(void *ptr, int N) {
    unsigned char mask = 128 >> (N-1);
    unsigned char valor = *((unsigned char*) ptr);
    valor = valor & mask;
    return valor > 0? '1' : '0';
}
```

La función `obtenerNbit` sólo puede retornar dos valores: el carácter '`0`' si el N -ésimo bit del byte almacenado en `ptr` está apagado, o '`1`' si, por el contrario, el bit está encendido.

La máscara se almacena en la variable `mask`, que empieza inicializada en 128 para aprovechar el hecho que este número posee un único bit 1 en su representación binaria, 10000000. A partir de este número podemos generar cualquier máscara desplazando su único bit 1 cierto número de espacios hacia la derecha. En general, una máscara con un bit encendido en la N -ésima posición se obtiene desplazando el único 1 de 10000000 $N-1$ espacios a la derecha, con `128 >> (N-1)`.

Los 8 bits almacenados en `ptr` son transferidos a la variable `valor`. Dado que `bit1 & bit2` siempre es 0 cuando alguno de los bits involucrados es 0, la expresión `valor & mask` produce un resultado en el que se preserva el N -ésimo bit de `valor` y los bits restantes se hacen 0's. Esto implica que si el N -ésimo bit de `valor` fuese 0, `valor & mask` será una secuencia de ocho 0's; de lo contrario, será una secuencia de siete 0's y un 1 (o sea, un valor positivo).

La variable `valor` es definida como `unsigned char` para evitar que se interprete el primer bit de `valor & mask` como bit de signo. Esto podría tener un impacto indeseado al evaluar la condición `valor > 0` en el operador ternario, en la última línea de la función `obtenerNbit`.

El código mostrado abajo imprime los 8 bits de una variable de tipo `char` cuyo valor es ingresado por el usuario:

```

#include <stdio.h>

char obtenerNbit(void *ptr, int n) {
    unsigned char mask = 128 >> (n-1);
    unsigned char valor = *((unsigned char*) ptr);
    valor = valor & mask;
    return valor > 0? '1' : '0';
}

void muestraByte(void *ptr) {
    for(int i=1; i<=8; i++)
        putchar(obtenerNbit(ptr, i));
}

int main(void) {
    char num;
    printf("Ingrese valor entre -128 y 127:");
    scanf("%d", &num);
    muestraByte(&num);
}

```

14. Números racionales

Defina un tipo de datos Racional con atributos num y den, junto a tres funciones que impriman, sumen y simplifiquen estos números.

Solución:

El tipo de datos Racional puede definirse de la siguiente forma:

```

typedef struct rational {
    int num;
    int den;
} Racional;

```

La impresión de un número racional debe considerar el caso cuando el denominador es 1. Ahí sólo se debe imprimir el numerador.

```

void imprime(Racional r) {
    if(r.den==1)
        printf("%d", r.num);
    else
        printf("%d/%d", r.num, r.den);
}

```

En cuanto a la simplificación de un número racional, esta se logra diviendo tanto el numerador como el denominador entre su máximo común divisor, de la siguiente forma:

```
Racional simplifica(Racional r) {  
    int factor = mcd(r.num, r.den);  
    r.num /= factor;  
    r.den /= factor;  
    return r;  
}
```

La función `mcd` es la misma que fue comentada en la sección de funciones recursivas. El número racional simplificado es retornado a la función llamadora mediante `return r`. El retorno de `r` a la función llamadora es necesario; si se omitiera, los cambios sobre `r.num` y `r.den` no podrían ser vistos fuera de `simplifica`. Recuerde que las estructuras son pasadas por valor, no por referencia.

La suma de dos racionales puede implementarse de la siguiente manera:

```
Racional suma(Racional r1, Racional r2) {  
    Racional r3;  
    r3.den = r1.den*r2.den;  
    r3.num = r1.num*r2.den + r1.den*r2.num;  
    return simplifica(r3);  
}
```

La suma almacenada en `r3` es simplificada y retornada al programa principal.

El lector puede escribir la función principal para validar la funcionalidad del código mostrado.

PROBLEMAS PROPUESTOS

- 1- Implemente una función que reciba un arreglo de números decimales y retorne la media aritmética, la media geométrica y la media armónica de los mismos.
- 2- Implemente una función que retorne la suma de dígitos del entero N pasado como parámetro.
- 3- Implemente una función que reciba una matriz A y determine si es triangular superior o no.
- 4- Implemente una función que reciba una matriz A y retorne su transpuesta, A^t .
- 5- Implemente una función que reciba dos matrices, A y B , y retorne su producto, $A \times B$.
- 6- Implemente una función que reciba una matriz A y determine si es idempotente, i.e. si $A^2 = A$.
- 7- Implemente una función que retorne el rango de la matriz pasada como parámetro.
- 8- Implemente una función que reciba una matriz A y retorne otra con las direcciones de memoria de cada elemento A .
- 9- Implemente una función que retorne el **producto interno** de dos puntos pasados como parámetros. Los puntos (x, y, z) deben implementarse como instancias de una estructura `Punto3D`, que deberá ser definida previamente.
- 10- Implemente una función que retorne el **producto vectorial** de dos puntos pasados como parámetros. Los puntos (x, y, z) deben ser implementados como instancias de una estructura `Punto3D`, que deberá ser definida previamente.
- 11- Defina el tipo de datos `Rectangulo`, con dos atributos de tipo `Punto2D`. Los valores de estos atributos deben corresponder a los vértices superior izquierdo e inferior derecho del rectángulo que se desea representar. Implemente una función que reciba dos `Rectangulo`'s y determine si se traslanan o no.
- 12- Implemente una función que reciba una matriz de caracteres (espacios en blanco y '#') y realice el borrado de la figura que contiene el punto (x, y) pasado como parámetro. Abajo se muestra lo que haría la función cuando es llamada con una matriz específica y parámetro $(3, 3)$: Borra toda la figura central. Si hubiese sido invocada con un parámetro $(0, 0)$, se habría borrado la L invertida que se encuentra en la esquina superior izquierda de la matriz.

#	#						
#				#	#		
		#	#	#			
		#	#			#	
	#	#				#	
					#	#	

Antes de invocar a la función

#	#						
#							
							#
							#
							#
						#	#

Después de la invocación con parámetros (3,3)

- 13- Escriba una función que reciba dos matrices de caracteres (espacios en blanco y #'s) y determine si las figuras representadas en dichas matrices encajan al ser sobreuestas, tal como en el gráfico adjunto, o no.

#	#	#	#	#	#	#	#
#	#						
#	#						
#	#	#	#				
#	#						
#	#						

		#	#	#	#	#	#
		#	#	#	#	#	#
				#	#	#	#
		#	#	#	#	#	#
		#	#	#	#	#	#

Dos matrices que sí encajan al ser sobreuestas

- 14- Implementar una función que reciba una matriz de caracteres (espacios en blanco y #'s) y retorne el número de bloques que componen la figura representada.

#	#	#	#				
#	#	#	#				
					#		
#	#	#			#	#	
#					#		
#							

Tres bloques de caracteres

		#					
		#					
#						#	
				#	#		
						#	
							#

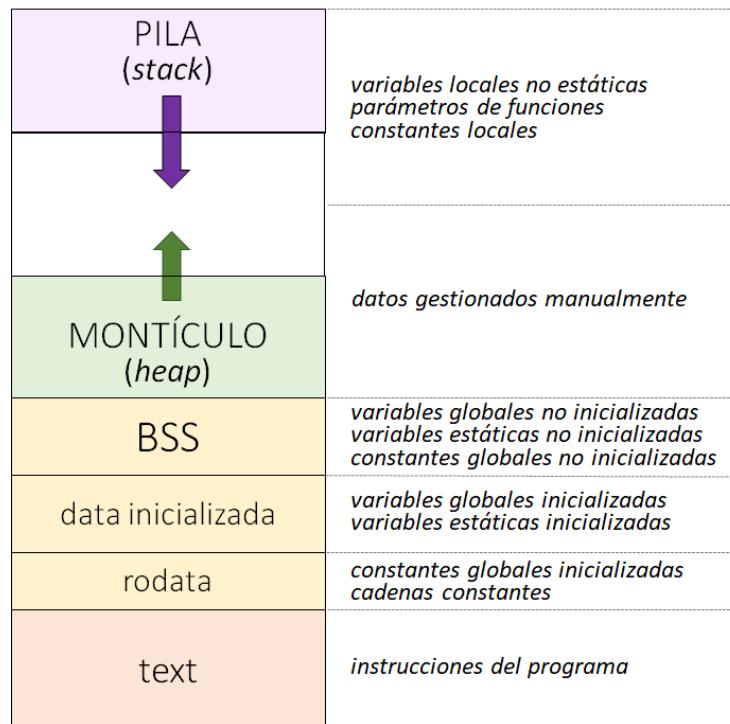
Dos bloques de caracteres

CAPÍTULO

8

MAPA DE MEMORIA

Cada vez que compilamos un programa este se transforma en un bloque humanamente ininteligible de 0's y 1's que luego, al momento de su ejecución, es cargado a la memoria del computador. En este bloque de 0's y 1's se encuentran codificados tanto los datos que manipula el programa como las instrucciones del mismo.



Mapa de memoria de un programa C

Para que el sistema pueda lidiar con semejante complejidad debe existir cierta organización que permita, por ejemplo, mantener las instrucciones separadas de los datos e, incluso, mantener los datos en diferentes regiones de la memoria, según sus características. Esto último es importante considerando que no todos los datos requieren el mismo tratamiento. Algunos necesitan permanecer más tiempo en memoria que otros, algunos requieren ser modificados y otros tienen que permanecer constantes, algunos aceptarán ser desalojados de las celdas que ocupan en memoria cuando el sistema operativo

lo considere oportuno mientras que otros exigirán mantenerse allí hasta que sea el mismo programa quien decida liberarlos.

La figura anterior se conoce como **mapa de memoria de un programa en C**. Es un modelo abstracto que nos da una idea de cómo están organizados los datos y las instrucciones de nuestros programas al momento de su ejecución. Vale aclarar que este modelo sólo es válido para los programas generados por compiladores del lenguaje C, no para programas hechos en otros lenguajes. Además debe tenerse en cuenta que una abstracción es simplemente eso –una abstracción. Dependiendo de cada sistema, la disposición física (real) de los segmentos podría presentar ligeras variaciones con respecto al modelo mostrado.

Cada variable que declaramos en nuestros programas ocupa cierto espacio en alguno de los segmentos mostrados. Ya hemos dicho que el espacio ocupado por una variable depende del su tipo (`int`, `float[]`, `char*`, etc.). Ahora añadiremos que el segmento al que una variable será confinada dependerá principalmente del **ámbito** donde ésta es declarada (local, global) y de los **modificadores** usados en su declaración (`static`, `const`, `auto`, etc.).

Es importante conocer las características de los segmentos que componen el mapa de memoria para así poder anticipar lo que podremos y no hacer con nuestras variables antes de declararlas, para saber lo que nos estará prohibido y lo que nos estará permitido, lo que debemos esperar del sistema y lo que tenemos que implementar nosotros mismos.

SEGMENTOS DE MEMORIA

MONTÍCULO (*heap*)

Por lo general, los datos de nuestros programas son almacenados en variables, pero no siempre tiene que ser así. Otra opción consiste en solicitar bloques de memoria de tamaño arbitrario dentro del segmento denominado montículo, para luego escribir nuestros datos manualmente en dicho bloque, en las celdas precisas, cuidando que no se traslapen unos con otros. Esta gestión manual de la memoria es técnicamente más complicada de implementar que simplemente asignar los datos a variables previamente declaradas; pero ofrece una ventaja. Uno podría extender varias veces, durante la ejecución del programa, el espacio de memoria solicitado inicialmente, lo cual sería imposible si sólo nos limitáramos a guardar nuestros datos en variables. Un arreglo de 100 elementos será siempre un arreglo de 100 elementos, nunca va a crecer en tiempo de ejecución. Usando el montículo, en cambio, uno podría inicialmente solicitar un bloque de 100 bytes, por decir algo, y extenderlo a un nuevo tamaño deseado en el momento oportuno, según las necesidades que vayan surgiendo durante la ejecución del programa. Esto permite un uso más eficiente de la memoria.

Para acceder al montículo se requiere utilizar una familia de funciones (`malloc`, `realloc`, `free`, etc.) que estudiaremos más adelante.

Los datos del montículo no tienen nombres; el programador debe recuperarlos a partir de sus direcciones.

Ninguna variable que declaremos se almacenará en el montículo. Allí sólo se guardan datos, sin nombre. Todas las variables que declaramos siempre terminarán ocupando espacio en la pila, el *BSS*, el *rodata* o el segmento de datos inicializados.

PILA (*stack*)

La pila es el segmento donde se almacenan datos que tienen un tiempo de vida corto, p.ej. variables locales. Dado que una variable local sólo es significativa dentro de la función en que es definida, no sería muy eficiente que siga ocupando espacio en la memoria luego que dicha función haya concluido su ejecución. Por ello, así como un mesero eficiente limpia el espacio que acaban de dejar unos comensales para acomodar otros nuevos, el sistema operativo está continuamente liberando de la pila el espacio ocupado por aquellas variables que van quedando obsoletas para reasignarlo a las variables de alguna función que haya comenzado a ejecutarse.

Otros datos que tampoco merecen ocupar espacio de memoria luego que finaliza la ejecución de las funciones donde fueron definidas son las constantes locales y los parámetros de función.

Mientras los datos almacenados en la pila son desalojados automáticamente por el sistema, los datos guardados en el montículo permanecen allí hasta que el mismo programa, mediante instrucciones explícitas (`free`), libere el espacio de memoria que ocupan.

Otro detalle a comentar aquí tiene que ver con las flechas mostradas en la figura anterior. Estas flechas indican las direcciones en las que crecerá cada segmento durante la ejecución de un programa. Ni la pila ni el montículo tienen tamaños fijos. La pila crece automáticamente cada vez que ocurre una llamada a una función (y entonces sus variables locales son cargadas en la memoria); el montículo crece cuando el programa solicita explícitamente más memoria. Por conveniencia, las direcciones de crecimiento de estos segmentos son opuestas. Si el montículo no está utilizando demasiado espacio, la pila puede tomar de esta área común el espacio que necesite, y viceversa. Además, las direcciones de crecimiento mostradas impiden que los datos de la pila y el montículo puedan llegar a sobrescribir el segmento de texto, donde residen las instrucciones del programa.

BSS

El *BSS*, cuyo nombre proviene de *Block Started by Symbol*, es el segmento donde se guardan los datos que no han sido explícitamente inicializados (p.ej. una variable global que ha sido declarada, pero a la que no se le ha asignado ningún valor al momento de la declaración). Todas las variables del *BSS* serán inicializadas implícitamente (con 0's en cada uno de sus bits) justo antes de iniciar la ejecución del programa.

SEGMENTO DE DATOS DE SÓLO LECTURA (*rodata*)

Aquí se almacenan las constantes del programa, aquellos datos que deben permanecer inalterados durante toda su ejecución. El nombre del segmento, *rodata*, proviene del inglés *read-only data* (data de sólo lectura).

SEGMENTO DE DATOS INICIALIZADOS

Aquí se almacenan, por ejemplo, las variables globales que han sido explícitamente inicializadas. Dado que estas deben ser accesibles durante todo el programa, independientemente de cual sea la función que se esté ejecutando, los datos almacenados en este segmento permanecerán visibles hasta que concluya el programa, bien a salvo de los continuos desalojos de memoria que suelen sufrir los datos alojados en la pila. Dentro de este segmento se almacenan también las variables estáticas (locales o globales) explícitamente inicializadas.

SEGMENTO DE TEXTO

El segmento de texto es el lugar de la memoria donde se almacenan las instrucciones del programa. Aunque los datos y las instrucciones de un programa están entremezclados en el código fuente, ambos permanecerán separados en la memoria. Todas las funciones que uno define en su programa son traducidas a bloques de 0's y 1's que terminan siendo alojados en este segmento.

MODIFICADORES

Además de conocer el tamaño en bytes que ocupa una variable es importante conocer también su tiempo de vida en memoria (automática o estática), su alcance (local o global) y su variabilidad (variable o constante). Estas características dependen de dos factores: (a) del lugar donde declaramos la variable en el programa, y (b) de los modificadores usados en su declaración.

A continuación discutiremos las características de distintas clases de variables y la sintaxis que debe utilizarse para declararlas.

VARIABLES AUTOMÁTICAS

Las variables automáticas son cargadas y desalojadas automáticamente de la pila cuando el flujo de control entra y sale, respectivamente, del bloque en el que estas son reconocibles. Los parámetros de función y las variables locales no estáticas son variables automáticas.

Para ilustrar el asunto mostraremos el siguiente código:

```

int hubo_error = 0;

double foo(int num)
{
    int azar = 1+rand()%num;
    return 1.0/azar;
}

int main(void) {
    double suma = 0;
    for(int cont=0; cont<10; cont++)
    {
        suma += foo(cont);
        printf("suma parcial=%f", suma);
    }
}

```

No se moleste en descifrar lo que hace este programa. Son poco más que instrucciones al azar que utilizaré para explicar cuáles variables (no) son automáticas y por qué (no) se consideran así.

La variable `hubo_error` no es automática porque no es una variable local; es una variable global, está definida afuera de todas las funciones. No se guardará en la pila al momento de la ejecución.

Analizando la función `foo`, tanto su parámetro de entrada `num` como la variable `azar` son automáticas. La primera es un parámetro de función; la segunda es una variable local no estática (su declaración no incluye el modificador `static`). Ambas subirán a la pila cada vez que se invoque a la función `foo` y permanecerán allí hasta que `foo` termine su ejecución y retorne el flujo de control a la función `main`. Dado que este caprichoso programa invoca 10 veces a `foo`, `num` y `azar` serán alojadas y liberadas 10 veces de la memoria, y no necesariamente ocuparán siempre las mismas celdas.

Dentro del `main`, las variables `suma` y `cont` son automáticas. Ambas son variables locales no estáticas. Sin embargo, el ciclo de vida de ambas será distinto. `suma` permanecerá en memoria durante toda la ejecución de `main`. En cambio, la variable `cont` sólo estará en memoria durante la ejecución del bucle `for`. Antes y después de ejecutar el bucle `for`, la variable `cont` no ocupará espacio en memoria.

Aunque hubiera sido en vano, se pudo haber añadido el modificador `auto` a todas las variables locales que acabamos de clasificar como automáticas (p.ej. se pudo haber declarado `auto double suma = 0`). Esto no habría cambiado nada, pues el modificador `auto` está implícito en todas las variables locales no estáticas.

Cuando una variable automática no es inicializada al momento de su declaración, su valor es indeterminado.

VARIABLES GLOBALES

Las variables globales son aquellas que se definen fuera de todas las funciones. Son reconocibles desde cualquier punto del programa, i.e. pueden ser leídas o actualizadas desde cualquier parte del código. Su tiempo de vida no depende de las múltiples llamadas a funciones que podrían estar ocurriendo en tiempo de ejecución.

Dependiendo de si son o no inicializadas al momento de su declaración, las variables globales podrían almacenarse en el *BSS* o en el segmento de datos inicializados. Discutamos el siguiente código:

```
int x=1000;
int y, z;

int main(void) {
    z=500;
}
```

Las tres variables de este inocuo programa son variables globales de tipo entero y, por lo tanto, cada una ocupará 4 bytes en la memoria. Podemos añadir que la variable `x`, por haber sido inicializada explícitamente (con el valor 1000), será alojada en el sector de datos inicializados. En cambio, las variables `y` y `z`, por no haber sido inicializadas explícitamente al momento de su declaración, irán a ocupar espacio en el *BSS*. No importa que la variable `z` haya asignada después de la declaración. Por no habersele dado un valor en la misma línea en que se declaró le corresponde el segmento *BSS*.

Justo antes de que empiece a ejecutarse la función principal, `main`, los elementos del *BSS* son inicializados a cero. Esto implica que si pudiéramos imprimir los valores de `x`, `y` y `z` en la última línea del `main` obtendríamos 1000, 0 y 500, respectivamente.

VARIABLES ESTÁTICAS

Las variables estáticas son declaradas usando el modificador `static`. Pueden ser locales o globales. En cada uno de estos casos el modificador `static` tiene una interpretación diferente.

VARIABLES ESTÁTICAS LOCALES

Llamaremos así a las variables estáticas que son declaradas dentro de una función. La característica más llamativa de estas variables es que pueden mantener su valor entre distintas llamadas a una función, tal como explicaremos con apoyo del ejemplo mostrado abajo.

<pre>void foo(void) { auto int i=1; printf("\n%d", i++); } int main(void) { foo(); // imprime 1 foo(); // imprime 1 foo(); // imprime 1 }</pre>	<pre>void foo(void) { static int i=1; printf("\n%d", i++); } int main(void) { foo(); // imprime 1 foo(); // imprime 2 foo(); // imprime 3 }</pre>
Cuando <code>i</code> es una variable automática	Cuando <code>i</code> es una variable estática

Los dos fragmentos de código mostrados son iguales, excepto por el modificador de la variable `int i=1`, declarada en la primera línea de la función `foo`.

En el código de la izquierda la variable `i` es **automática**. En cada llamada a `foo` su valor es reinicializado, `i=1`. Es imposible recuperar el valor de `i` después de una llamada a `foo` porque es justamente en ese momento cuando la variable `i` deja de existir. Una posterior llamada a `foo` volverá a crear una «nueva» variable `i`, probablemente en una nueva dirección de memoria, pero el valor de la variable `i` que expiró al concluir la llamada anterior ya no podrá ser recuperado.

En el código de la derecha la variable `i` es **estática**. Esto significa que ya no «morirá» luego que se termine la ejecución de `foo`. Ahora la variable `i` descansa a salvo en el segmento de datos inicializados y, por lo tanto, en nada le afecta que el sistema se pase sobreescribiendo la pila todo el tiempo. El valor de la variable estática `i` permanece en la memoria luego de cada llamada a `foo` y puede ser recuperado en la siguiente llamada.

Siguiendo con el código de la derecha, la asignación `static int i=1` sólo será tomado en cuenta una vez, en la primera llamada a `foo`. Las llamadas posteriores ignorarán la inicialización `i=1`. Con el modificador `static` ya no hay necesidad de crear un «nuevo `i`» en cada llamada a `foo`. La `i` estática que se crea en la primera llamada permanecerá en el segmento de datos durante todo el programa.

Las variables estáticas que no son explícitamente inicializadas se guardan en el `BSS` y son automáticamente inicializadas en cero al inicio del programa.

VARIABLES ESTÁTICAS GLOBALES

Llamaremos así a las variables estáticas que son declaradas fuera de todas las funciones. Este tipo de declaración sólo tiene sentido cuando trabajamos con programas multiarchivos, programas cuya funcionalidad es especificada en diferentes archivos, cuyos código fuente terminan integrándose en un único ejecutable gracias al enlazador (*linker*).

En estos casos el modificador `static` sirve para indicar que una variable global sólo debe ser reconocida dentro de su propio archivo, no desde otros archivos del programa. Las variables estáticas globales que han sido explícitamente inicializadas se almacenan en el segmento de datos inicializados. El resto ocupará espacio dentro del *BSS*.

Para ilustrar lo dicho discutamos el siguiente programa compuesto de dos archivos. Si enlazamos los códigos `prog1.c` y `prog2.c` (haciendo `gcc prog1.c prog2.c`), el archivo ejecutable resultante imprimirá dos veces cero durante su ejecución. Esto porque al declarar la variable global `gb` como `static` en el archivo `prog2.c`, esta no podrá ser reconocida desde `prog1.c`. La variable `gb` de `prog1.c` y la variable `gb` de `prog2.c` no tienen ninguna relación entre ellas y ocuparán distintas posiciones de memoria. La `gb` de `prog1.c` irá al segmento de datos inicializados; la de `prog2.c`, al *BSS*.

<pre>int gb = 100; void foo(void); int main(void){ foo(); // imprime 0 gb = 200; foo(); // imprime 0 }</pre>	<pre>static int gb; void foo(void) { printf("%d", gb); }</pre>
<code>prog1.c</code>	<code>prog2.c</code>
Programa multiarchivo que imprime 0 0 como salida	

Algo diferente ocurriría si el modificador `static` fuese omitido o reemplazado por `extern`, tal como en el siguiente programa. En este nuevo caso habría una única variable global `gb` en el código enlazado, la que fue definida en `prog1.c` (`extern` indica que `gb` ya fue declarada en otro archivo). En este caso, la salida del programa sería 100, 200. La variable `gb` impresa desde `prog2.c` sería la misma que fue declarada en `prog1.c`. Esta se almacena en el segmento de datos inicializados.

<pre>int gb = 100; void foo(void); int main(void){ foo(); // imprime 100 gb = 200; foo(); // imprime 200 }</pre>	<pre>extern int gb; void foo(void) { printf("%d", gb); }</pre>
<code>prog1.c</code>	<code>prog2.c</code>
Programa multiarchivo que imprime 100 200 como salida	

CONSTANTES

Cuando se añade el modificador `const` a una declaración de variable el valor de ésta quedará fijo y ya no podrá ser actualizado durante el resto del programa.

Las constantes globales inicializadas explícitamente, p.ej. `gravedad`, se almacenan en el segmento de datos de sólo lectura, `rodata`; las constantes globales no inicializadas, p.ej. `nulo`, se almacenarán en el `BSS`; las constantes locales, ya sea que estén inicializadas o no, p.ej. `r2` y `x`, se almacenarán en la pila.

```
const double gravedad = 9.81;
const int nulo;

int main(void) {
    const double r2 = 1.4142;
    const int x;
}
```

La declaración de una constante local no inicializada en realidad no tendría mucho sentido, puesto que tendrá un valor indeterminado que nunca podrá ser cambiado.

Por estar dentro del `BSS`, la variable global `nulo` será inicializada a cero justo antes de que se empiece a ejecutar la función principal, `main`.

CADENAS

Cada vez que se asigna un texto encomillado a un puntero de tipo `char*`, el texto entre comillas se almacena en el segmento de sólo lectura, `rodata`. Por ejemplo, una declaración como:

```
int main(void) {
    char *name= "Anita";
}
```

hace que (1) los caracteres A, n, i, t, a, '\0' se copien en la sección `rodata`. Además, (2) se ocupará 8 bytes en la pila para almacenar el puntero `name` que apuntará a la cadena "Anita". La variable `name` irá a la pila por ser automática, i.e. variable local no estática.

Tal como hemos visto hasta aquí, para que podamos dirigir nuestros datos hacia la pila, el `BSS`, el `rodata` o el segmento de datos inicializados, según nuestra conveniencia, basta con saber cómo y dónde declarar una variable, si se debe utilizar tal o cual modificador, y si debe declararse dentro de alguna función o fuera de todas ellas.

CAPÍTULO

9

PUNTEROS A FUNCIONES

Así como es posible almacenar la dirección de una variable, de un arreglo de variables y de una estructura de datos, también es posible guardar la dirección de una función. Los punteros que almacenan estas direcciones se denominan punteros a funciones.

La declaración de un puntero a función es más intrincada que la de otros tipos de punteros. Depende de dos factores: (1) de los tipos de parámetros que recibe la función, y (2) del tipo de dato que retorna. Por ejemplo, un puntero `ptr` definido como:

```
double (*ptr)(double, double);
```

sólo podrá apuntar a funciones que reciban dos parámetros de tipo `double` y que retornen un valor `double`. Así, `ptr` podría apuntar a funciones como las siguientes:

```
double suma(double x, double y) {return x+y;}  
double resta(double x, double y) {return x-y;}
```

pero no a funciones como:

```
double invierte(int x) { return 1.0/x; }  
void saluda(void) { puts("Hola"); }
```

Para asignarle un valor a un puntero a función se puede utilizar la siguiente sintaxis:

```
ptr = &suma;
```

Dado que el compilador interpreta los nombres de funciones como las direcciones de estas, la expresión anterior es equivalente a `ptr = suma` (sin el ampersand, `&`). Una asignación como:

```
ptr = &invierte;
```

haría que el compilador lance una advertencia por tipos de punteros incompatibles. A pesar de que la función `invierte` retorna un dato de tipo `double`, su definición dice que recibe un parámetro de tipo

`int`, y no dos parámetros de tipo `double`, tal como exige la definición de `ptr`. Si uno quisiera guardar la (dirección de la) función `invierte` se tendría que definir otro puntero, `ptr2`:

```
double (*ptr2)(int) = &invierte;
```

Y, análogamente, para la función `saluda` se tendría que declarar algo como:

```
void (*ptr3)(void) = &saluda;
```

Un puntero a función puede ser utilizado para invocar a la función a la que apunta. Para el código que venimos mostrando, la invocación `suma(3,5)` podría hacerse del siguiente modo:

```
double resultado = (*ptr)(3,5);
```

También era posible omitir el operador de desreferencia y escribir simplemente `ptr(3,5)`. En ambos casos la variable `resultado` habría tomado el valor de 8, la suma de 3 y 5.

Algunas de las posibles formas de invocar las funciones `invierte` y `saluda` a través de los punteros `ptr2` y `ptr3`, respectivamente, son las siguientes:

```
ptr3();  
printf("\nLa inversa de 10 es %f", ptr2(10));
```

En el caso de `ptr3()` no hay necesidad de pasar parámetros ni de esperar uno de vuelta. Está invocación simplemente imprimirá "Hola" en la pantalla. Por otro lado, la invocación de `ptr2(10)` retornará la inversa de 10, la cual, sin tener que ser almacenada en una variable, irá directamente a la pantalla, a ser impresa como número flotante, `%f`.

No está demás mencionar que no habría ningún problema en reasignar el puntero `ptr` siempre que se le haga apuntar a funciones del tipo `double (*)(double, double)`. Según esto, la instrucción:

```
ptr = resta;
```

es correcta y no ocasiona ningún problema. Después de todo, un puntero es una variable y, como su nombre sugiere, puede variar: en un momento podría estar apuntando a una dirección y milisegundos más tarde (o líneas más abajo, según la perspectiva del lector) podría apuntar a otra nueva.

No hay que perder de vista que luego de la reasignación de `ptr`, la instrucción `ptr(3,5)` que antes nos devolvió 8 ahora nos retornaría un -2; es decir, ya no la suma $3+5$ sino la resta $3-5$.

TAMAÑO DE UNA FUNCIÓN

Durante la compilación, las instrucciones con que definimos una función son convertidas a lenguaje de máquina, a un bloque de 0's y 1's cuyo contenido es imposible de descifrar para cualquier humano. Durante la ejecución, este bloque de instrucciones es cargado a la memoria para que el sistema pueda leerlo desde allí y ejecutarlo instrucción por instrucción. Esta es una de las ideas brillantes que hay detrás del computador y que usualmente olvidamos o dejamos de apreciar: El mismo espacio que se utiliza para almacenar datos codificados –me refiero a la memoria– es también utilizado para almacenar instrucciones ejecutables codificadas. Al margen de cualquier sentimentalismo, lo que quiero decir es que, al igual que las variables, las funciones también ocupan espacio dentro de la memoria en tiempo de ejecución. Pero, a diferencia de las variables, el espacio que ocupan las instrucciones de una función no puede ser determinado con la función `sizeof`.

La expresión `sizeof(*ptr)` siempre retornará 1. Esto no significa que cada función que usted implementa tan esforzadamente puede ser representada internamente como una humillante hilera de 8 bits (1 byte). Simplemente significa que no se puede usar el operador `sizeof` para determinar el tamaño de una función.

Si usted prefiriese que el compilador le advierta sobre el uso de una expresión sin sentido como `sizeof(*ptr)`, puede compilar su archivo fuente con: `gcc -pedantic micódigo.c`.

Por otro lado, la expresión `sizeof(ptr)` sí funciona correctamente, siempre retorna 8, el tamaño en bytes de un puntero.

ARITMÉTICA DE PUNTEROS A FUNCIÓN

La aritmética de punteros a funciones no tiene mucho sentido. La expresión `ptr+1` será interpretada como la dirección del siguiente byte después de `ptr`. La compilación con la cláusula `-pedantic` también servirá para que el compilador le advierta del sinsentido que implica utilizar aritmética de punteros a funciones en su programa.

RETROLLAMADAS (*callbacks*)

Una de las ventajas de declarar punteros a funciones es que podemos pasar dichos punteros como argumentos a otras funciones. Luego, desde el interior de la función invocada, es posible llamar a la función que fue pasada como argumento.

Más concretamente, cuando un puntero `ptr` que apunta a una función `A` es pasado como argumento a una función `B`, p.ej. haciendo `B(ptr)`, la funcionalidad de `A` puede ser invocada desde dentro de `B`; bastaría con llamar a `ptr` junto a la lista de parámetros que pudiera necesitar. En este caso se dice que `A` es una **función de retrollamada** (*callback function*, en inglés), una función pasada como argumento para ser invocada desde dentro de otra función.

La capacidad de pasar (punteros a) funciones a otras funciones permite desarrollar código de alto nivel de abstracción. Ahora sería posible, por ejemplo, crear una función `Integral` que acepte llamadas como:

```
Integral(f, 0, 1)  
Integral(g, 1, 5)  
Integral(h, -1, 1)
```

y que retorne las integrales definidas de `f` en el intervalo $[0, 1]$, de `g` en el intervalo $[1, 5]$ y de `h` en el intervalo $[-1, 1]$, respectivamente. El hecho de poder pasar las funciones `f`, `g` y `h` como argumentos nos permite implementar una función `Integral` bastante genérica, definida a un nivel de abstracción tan alto que podría trabajar correctamente, independientemente de cual sea la función a integrar. Para definir una función como `Integral`, esta debería admitir un puntero a función como primer parámetro. Para ello, su cabecera debe definirse así:

```
double Integral(double (*ptr)(double), double a, double b)
```

Con esta declaración, cualquier función continua, real, de variable real, que pueda ser implementada como tipo `double (*)(double)`, podría ser pasada a `Integral`.

Para ver una posible implementación de esta función revise la sección de ejercicios.

ARREGLO DE PUNTEROS A FUNCIONES

Varios punteros a funciones pueden ser almacenados en un mismo arreglo, tal como se muestra abajo:

```
double (*ptr1)(double, double) = suma;  
double (*ptr2)(double, double) = resta;  
double (*aptrs[2])(double, double) = {ptr1, ptr2};
```

`aptrs` es un arreglo de dos punteros a funciones que reciben `(double, double)` como parámetros de entrada y retornan un dato de tipo `double`. Aunque el código anterior es correcto, en realidad no había necesidad de utilizar `ptr1` y `ptr2` en la declaración de `aptrs`. También se pudo utilizar directamente las direcciones de `suma` y `resta` haciendo:

```
double (*aptrs[2])(double, double) = {suma, resta};
```

Recuerde que el nombre de una función se interpreta como la dirección del primer byte que ocupa su código en memoria.

Un arreglo de (punteros a) funciones puede ser utilizado, por ejemplo, para gestionar un menú de opciones. Cada opción disponible podría coincidir con los subíndices de un arreglo de funciones, de modo que cuando el usuario elija una opción específica se active directamente la funcionalidad asociada a dicho subíndice, sin necesidad de utilizar sentencias condicionales. Otro uso podría ser para responder a las sentencias ingresadas en una línea de comando, tal como se muestra abajo:

<pre> int x, y, rpta; char ope; double (*aptrs[2])(double, double) = { suma, resta }; while (1) { printf(">>"); scanf("%d %c %d", &x, &ope, &y); switch(ope) { case '+': rpta = aptrs[0](x,y); break; case '-': rpta = aptrs[1](x,y); break; default: puts("Operación desconocida"); continue; } printf("Respuesta = %d", rpta); } </pre>	<pre> User@DESKTOP-ETMGBI ~ \$./a >>3 + 6 Respuesta=9 >>24+9 Respuesta=33 >>13-7 Respuesta=6 >>4*8 Operación desconocida >>-12 - 4 Respuesta=-16 </pre>
--	--

El programa empieza mostrando un indicador de línea de comandos, `>>`, a través del cual el usuario ingresará una operación aritmética con dos operandos. El primer operando, el operador y el segundo operando son capturados en las variables `x`, `ope` e `y`, con `scanf ("%d %c %d", &x, &ope, &y)`.

La operación solicitada, `ope`, sirve para que el programa elija una función del arreglo `aptrs`, `suma` ó `resta`, la cual será aplicada a los operandos almacenados en `x` y `y`.

Tal como está escrito el código, incluso aunque se cambien los nombres de las funciones `suma` y `resta`, para que todo siga funcionando correctamente sólo tendríamos que cambiar la declaración de `aptrs`, una sola línea de código.

Dejamos al lector que complete el código anterior hasta obtener una calculadora aritmética básica.

CONVERSIÓN DE TIPOS ENTRE FUNCIONES

En la sección anterior vimos como declarar arreglos de funciones que tienen la misma firma (i.e. el mismo número y tipo de parámetros y el mismo tipo de retorno). Ahora hablaremos de arreglos de funciones de firma distinta. Para ello necesitamos utilizar punteros de tipo genérico, `void*`. Así podemos almacenar en un mismo arreglo funciones como `suma`, `invierte` y `saludo`, con el siguiente código:

```
void *aptrs[3] = { suma, invierte, saludo};
```

Es cierto que esta sintaxis es mucho más simple que la mostrada en la sección anterior. Pero en este caso tendremos que lidiar otro problema. La invocación `(*aptrs[0])(3,5)`, por citar un ejemplo, no conduciría a la llamada de `suma(3,5)` sino a un mensaje de error conocido: «No se puede desreferenciar un puntero genérico». Para invocar las funciones almacenadas en `aptrs` se debe utilizar conversiones de tipos (*casting*), tal como se muestra a continuación:

```
double a = ((double (*) (double, double))aptrs[0])(3, 5);
```

Esta sentencia no causa errores ni advertencias. La expresión `(double (*) (double, double))` a la izquierda de `aptrs[0]` indica que la dirección genérica almacenada en `aptrs[0]` es la dirección de una función; en particular, de una función que recibe `(double, double)` y retorna `double`. Solo así es posible invocar a la función `suma` desde `aptrs[0]`; en nuestro caso, con parámetros 3 y 5.

El paso de los argumentos `(3,5)` debe realizarse después de especificar el tipo de `aptrs[0]`. Por ello hay un par de paréntesis encerrando toda la expresión que precede a `(3,5)`, para garantizar el orden de evaluación deseado. En la asignación anterior la variable `a` recibirá el resultado de `suma(3,5)`.

Siguiendo con el mismo razonamiento, ahora es posible invocar y almacenar el resultado de la función `invierte(10)` en la variable `b` haciendo:

```
double b = ((double (*) (int))aptrs[1])(10);
```

Y también se puede escribir un saludo en pantalla con la instrucción:

```
((void (*) (void))aptrs[2])();
```

RETORNO DE UN PUNTERO A FUNCIÓN

Así como una función puede recibir funciones también puede devolverlas como valor de retorno. Con esta capacidad podríamos, por ejemplo, definir una función que retorne la derivada de otra pasada como parámetro. Sin ánimos de complicar el asunto, reduzcamos este problema de la derivada a unas pocas funciones trigonométricas. El código, por ahora incompleto, quedaría más o menos así:

```
[?] derivada (double (*f)(double)) {
    if(f==sin) return cos;
    if(f==cos) return dcos;
    if(f==tan) return dtan;
}
```

Las funciones `sin`, `cos` y `tan` se encuentran ya declaradas en el archivo `math.h`; las funciones `dcos` y `dtan` deben ser implementadas por uno mismo. Dado que `sin`, `cos` y `tan` están definidas para recibir y retornar un dato `double`, definiremos `dcos` y `dtan` de la misma manera. No es tan difícil si uno recuerda las fórmulas de las derivadas del seno y la tangente aprendidas en la escuela; cada una puede definirse en una línea de la siguiente forma:

```

double dcos(double x) { return -sin(x); }
double dtan(double x) { return 1/pow(cos(x), 2); }

```

En cuanto a la función `derivada`, ésta primero identifica la función recibida como parámetro y luego retorna otra función, su derivada correspondiente. Note que es válido comparar funciones con el operador `==`. Esto no debería sorprenderlo puesto que `f`, `sin`, `cos` y `tan` son punteros y, como ya hemos visto, los punteros pueden compararse sin ningún problema.

Es fácil equivocarse e intentar definir `dtan` para que retorne `return 1/cos*cos`. Esto sería absurdo, ya que la multiplicación de punteros, `cos*cos`, no está definida.

Ahora falta lo más difícil: definir el tipo de retorno de `derivada`. ¿Cómo podemos especificar que `derivada` va a retornar un puntero a función? La sintaxis es un poco intrincada. Por ello, sin temor a perder la formalidad, me veo obligado a revelar el método de la «estrella solitaria» que suelo utilizar en clases con mis alumnos, diría que con relativo éxito.

El método es el siguiente: Primero escriba como tipo de retorno el tipo de puntero que se piensa devolver, es decir `double (*) (double)`. Nuestra definición quedaría como algo así:

```
double (*) (double) derivada (double (*f) (double))
```

Ahora falta el segundo y definitivo paso. Identifique a la estrella solitaria, `(*)`,—siempre habrá una cada vez que quiera retornar funciones— y acompañela con todo el resto de la cabecera que no pertenezca al tipo de retorno, i.e. con el código que está a la derecha de `double (*) (double)`. Luego de una simple operación de cortar-pegar, la cabecera de la función `derivada` quedaría correctamente declarada de la siguiente forma:

```
double (*derivada (double (*f) (double))) (double)
```

El cuerpo de la función sería el mismo que ya discutimos anteriormente.

En cuanto a la invocación de `derivada`, uno podría hacer una llamada como:

```
derivada(sin) (M_PI_4)
```

En este caso primero se ejecutará `derivada(sin)`, lo que retornará la función `cos`. Luego, esta función será evaluada en $\pi/4$, dandonos como resultado final 0.707107, el coseno de $\pi/4$. Este resultado podría capturarse en una variable `double` o ser directamente impresa con el especificador de formato `%f`, por mencionar un par de posibilidades. La constante `M_PI_4` está incluida en `math.h`.

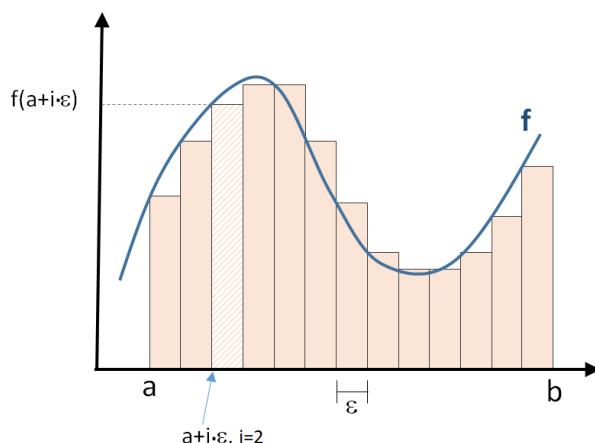
PROBLEMAS RESUELTOS

1. Recordando las lecciones de Cálculo Integral

Implemente la función Riemann(f , a , b), que retorna la integral definida de la función f en el intervalo $[a, b]$.

Solución:

De nuestras clases de Cálculo recordamos que la integral de f en $[a, b]$ es el área bajo la curva f en el intervalo $[a, b]$. Sabemos además que esta área puede ser aproximada como la suma de las áreas de varios rectángulos de ancho infinitesimal, tal como se ve abajo:



Lo que haremos primero será definir este ancho infinitesimal como $\text{eps} = (b-a) / N$. Así, para aproximar la integral tendremos que sumar el área de N rectángulos, cuyas bases tienen el mismo ancho. Cuanto más grande sea N más pequeño será eps y mejor será nuestra aproximación de la integral como suma de Riemann.

Una variable `Area` servirá para acumular las áreas de los escuálidos rectángulos que se muestran en la imagen. En general, el área del i -ésimo rectángulo puede ser calculada como el producto de su base, eps , multiplicado por su altura, $f(a+i\cdot\text{eps})$. Mientras la base, eps , es la misma para todos los rectángulos, sus alturas $f(a+i\cdot\text{eps})$ dependen de la variable i ; o sea que varían de un rectángulo a otro.

Dado que necesariamente hay N rectángulos sobre $[a, b]$, las áreas $\text{eps} \cdot f(a+i\cdot\text{eps})$ deben ser calculadas y acumuladas para $i=0, 1, 2, \dots, N-1$.

La función Riemann quedaría como sigue:

```
#include <stdio.h>

double Riemann(double (*f)(double), double a, double b) {
    double N=10000;
    double eps = (b-a)/N;
    double Area = 0.0;
    for(int i=0; i<N; i++)
        Area += eps*f(a+eps*i);
    return Area;
}
```

Esta función aproxima la integral definida de f en $[a, b]$ como una suma de $N=10000$ rectángulos del mismo ancho. N también pudo ser declarado como parámetro de la función.

Podemos definir arbitrariamente una función $g(x) = 1+x^2$ para evaluar la función Riemann, tal como se muestra abajo. La salida de $Riemann(g, 0, 5)$ será 46.660417, una buena aproximación de la integral $\int g(x) dx = \int 1+x^2 dx$ sobre $[0, 5]$, cuyo valor exacto, calculado a mano, sería el número decimal periódico puro 46.666666...

```
double g(double x) { return 1+x*x; }

int main(void) {
    printf("%f=", Riemann(g, 0, 5));
}
```

2. Una cadena de llamadas

Implemente la función $F(f, n, x)$, que retorne el valor de la siguiente expresión:

$f(f(f(\dots f(x) \dots)))$, donde f se aplica n veces

Solución:

Pensando recursivamente diríamos que aplicar n veces la función f sobre x , i.e. $F(f, n, x)$, sería lo mismo que aplicar $n-1$ veces f sobre $f(x)$, i.e. $F(f, n-1, f(x))$.

El problema es trivial cuando $n=1$. En ese caso la función f se aplica una sola vez, por lo que se retornaría directamente $f(x)$.

El código pedido es el siguiente:

```
double F(double (*f)(double), int n, double x) {
    if(n==1)
        return f(x);
    else
        return F(f, n-1, f(x));
}
```

3. Wronskiano

Se le pide implementar una función Wronskiano que reciba los siguientes datos:

- Una matriz W de dimensiones $N \times M$,
- Un arreglo de M elementos $\{a_0, a_1, a_2, \dots, a_{M-1}\}$, y
- Un arreglo de punteros a las funciones $\{f, f', f'', \dots, f^{(N-2)}\}$

La función debe modificar la matriz recibida según el patrón que se muestra abajo para el caso particular de $N=4$ y $M=5$.

a_0	a_1	a_2	a_3	a_4
$f(a_0)$	$f(a_1)$	$f(a_2)$	$f(a_3)$	$f(a_4)$
$f'(a_0)$	$f'(a_1)$	$f'(a_2)$	$f'(a_3)$	$f'(a_4)$
$f''(a_0)$	$f''(a_1)$	$f''(a_2)$	$f''(a_3)$	$f''(a_4)$

Solución:

Como las matrices siempre son pasadas por referencia la función Wronskiano no necesita retornar nada, le bastará con modificar la matriz recibida desde la función llamadora. El parámetro para recibir la matriz puede declararse como `double W[N][M]`, con N y M previamente declaradas.

El parámetro que recibirá el arreglo $\{a_0, a_1, a_2, \dots, a_{M-1}\}$ será `double as[M]`. El que recibirá los punteros a funciones $\{f, f', f'', \dots, f^{(N-2)}\}$ será declarado como `double (*fs[N-1])(double)`.

Así, la extensa cabecera de Wronskiano sería:

```
void Wronskiano ( int N, int M, double W[N][M],
                  double as[M],
                  double(*fs[N-1])(double) )
```

La función comenzará asignando la primera fila de W con los elementos de `as`, haciendo:

```
W[0][j] = as[j] , j=0, 1, ..., M-1
```

Luego se completará el resto de la matriz. Observe que cada elemento es la aplicación de alguna función del arreglo `fs` sobre algún elemento de la primera fila. En general:

```
W[i][j] = fs[i-1](as[j]) , i=1, 2, ..., N-1; j=0, 1, ..., M-1
```

El código completo se muestra a continuación:

```
#include <stdio.h>

void Wronskiano(int N, int M, double W[N][M], double as[M],
                 double (*fs[N-1])(double))
{
    for(int j=0; j<M; j++)
        W[0][j] = as[j];

    for(int i=1; i<N; i++)
        for(int j=0; j<M; j++)
            W[i][j] = fs[i-1](as[j]);
}

double f1(double x) { return x*x*x; }
double f2(double x) { return 3*x*x; }
double f3(double x) { return 6*x; }

int main(void) {
    const int nfils = 4;
    const int ncols = 5;
    double W[nfils][ncols];
    double as[] = {1, 2, 3, 4, 5};
    double (*fs[])(double) = {f1, f2, f3};

    Wronskiano(nfils, ncols, W, as, fs);

    for(int i=0; i<nfils; i++) { // imprime la matriz
        for(int j=0; j<ncols; j++)
            printf("%.2f\t", W[i][j]);
        printf("\n");
    }
}
```

El primer bucle de `Wronskiano` sirve para asignar los elementos de la primera fila ($i=0$). El segundo se encarga de las filas siguientes, $i=1, 2, \dots, N-1$. A los elementos de la i -ésima fila siempre se les aplica la $(i-1)$ -ésima función.

Dentro del `main` se ha evaluado la función `Wronskiano` con las derivadas de $f(x)=x^3$ y el arreglo $\{1, 2, 3, 4, 5\}$ como primera fila. La salida del programa es la matriz:

```
$ ./wronskiano
1.00    2.00    3.00    4.00    5.00
1.00    8.00   27.00   64.00  125.00
3.00   12.00   27.00   48.00   75.00
6.00   12.00   18.00   24.00   30.00
```

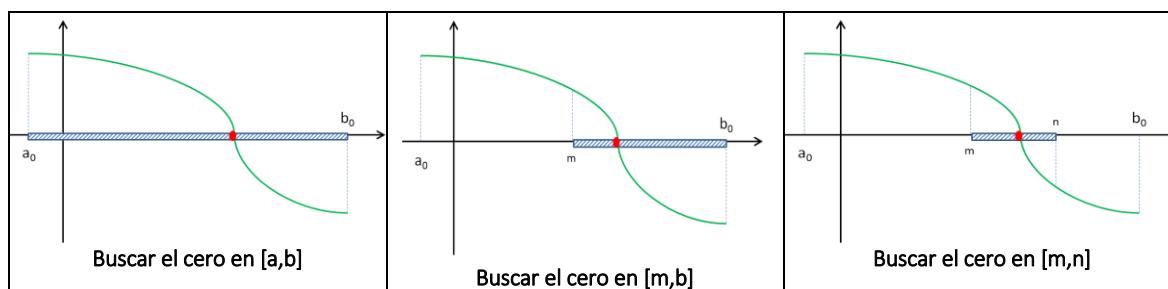
4. Método de la bisección (y otra aproximación de π)

Implemente una función que reciba (un puntero a) una función f y un intervalo (a, b) , y que retorne el valor x donde se cumple que $f(x)=0$; o sea, el cero de f en (a, b) . Asuma que sólo existe un valor que satisface tal condición.

Solución:

Aplicaremos la estrategia Divide y Vencerás. Dado que hay un único cero en (a, b) , este tiene que estar necesariamente en la mitad derecha o en la mitad izquierda de dicho intervalo. De saberlo, podríamos reducir la búsqueda inicial sobre (a, b) a otra sobre (a, m) ó (m, b) , siendo m el punto medio de (a, b) . Este proceso podría repetirse una y otra vez.

En la secuencia de gráficos mostrada abajo, la búsqueda inicial sobre el intervalo (a, b) se reduce primero a una búsqueda sobre (m, b) , su subintervalo derecho, para luego volver a reducirse a una búsqueda sobre (m, n) , la mitad izquierda de (m, b) . Se podría continuar así, buscando en espacios cada vez más pequeños, reduciendo en cada paso el intervalo de búsqueda a la mitad.



Podríamos considerar como condición de parada el momento en que el intervalo de búsqueda es pequeño. En dicho caso, una aproximación del cero buscado sería el centro de ese microintervalo.

Para determinar si el cero buscado está a la derecha o a la izquierda del punto medio m de (a, b) podemos aplicar el siguiente razonamiento: Si $f(a) * f(m) < 0$, eso significa que $f(a)$ y $f(m)$ tienen signos diferentes; o sea, que la curva de f está por encima del eje X en un extremo de (a, m) y por debajo en el otro. Esto implica que la curva necesariamente atraviesa el eje X en algún punto de (a, m) y, por lo tanto, allí es adonde deberíamos redirigir la búsqueda del cero. Análogamente, $f(m) * f(b) < 0$ nos dice que el cero se encuentra en el subintervalo (m, b) .

```
double HallaCero(double (*f)(double), double a, double b) {
    double m = (a+b)/2;
    if(b-a < 0.000001)
        return m;
    else {
        if(f(m)*f(a)<0)
            return HallaCero(f, a, m);
        else if(f(m)*f(b)<0)
            return HallaCero(f, m, b);
        else if(f(m)==0)
            return m;
    }
}
```

La invocación `HallaCero(f, a, b)` retorna el cero de `f` en `(a, b)`. Las llamadas recursivas a `HallaCero` se detienen cuando el intervalo de búsqueda se reduce a una longitud menor que un millonésimo, $b-a < 0.000001$. Este nivel de tolerancia es arbitrario.

El código mostrado ha considerado un caso adicional, que no fue comentado arriba. Antes mencionamos las condiciones para saber si el cero buscado estaba a la derecha de `m` o a la izquierda de `m`; pero el programa también ha considerado el caso, poco probable, de que el cero no esté ni a un lado ni a otro, sino que sea exactamente `m`, i.e. $f(m) == 0$.

Para evaluar la función `HallaCero` podemos incluir el archivo `math.h` y hacer:

```
int main(void) {
    double x = HallaCero(sin, 0, 5);
    printf("%.6f", x);
}
```

Este código nos permite hallar el cero de la función seno, en `(0, 5)`; o sea, el valor de `x` donde se cumple que: $\sin(x) = 0$. Se sabe que dicho valor es igual a π .

La salida del programa arroja `3.141592`, que coincide en seis dígitos decimales con el valor real de π . Si cambia la precisión usada en `HallaCero` a `0.00000000000001` (10^{-14}) y el formato a `% .14f`, el programa arroja `3.14159265358980`, cuyos primeros doce dígitos decimales coinciden con los de π .

5. Polimorfismo

El polimorfismo es uno de los conceptos pilares de la programación orientada a objetos (POO). Sigue involucrar una clase base y varias clases derivadas de esta, de modo que una función definida en la clase base puede ser implementada de distintas formas en las clases derivadas. A pesar de que C no suscribe el paradigma orientado a objetos, es posible emular el polimorfismo (de manera primitiva); es decir, se puede definir una función que posea diferentes implementaciones.

Solución:

Un ejemplo muy común que se usa para ilustrar el polimorfismo de lenguajes OO como Java y C++ consiste en definir una clase base, p.ej. `Animal`, y varias clases derivadas, p.ej. `Perro`, `Gato`, `Pato`, etc. La clase base suele contener un método, p.ej. `haz_ruido()`, que se implementa de distintas formas en las clases derivadas, p.ej. para imprimir `guau guau`, `miau` o `cua cua`, respectivamente.

Aunque no existe el concepto de clases ni herencia en C, la funcionalidad anterior puede emularse implementando estructuras que posean punteros a funciones como atributos.

En el siguiente código, la estructura `struct animal` posee dos atributos: (1) una cadena, que almacena el nombre de un animal, y (2) un puntero a función, donde se guarda la función que hará ladrar/maullar/graznar a cada variable de tipo `struct animal`.

El polimorfismo puede considerarse emulado cuando al llamar a la misma función desde distintas variables (objetos, en términos de POO) se invocan distintas funcionalidades.

```
#include <stdio.h>

struct animal {
    char *nombre;
    char *(*sonido) (void);
};

char* guau(void) { return "guau guau"; }
char* miau(void) { return "miau miau"; }
char* cua(void) { return "cua cua"; }

void hacer_sonido(struct animal *pa) {
    printf("%s hace %s\n", pa->nombre, (pa->sonido) ());
}

int main(void) {
    struct animal perro = {"Fido", guau};
    struct animal gato = {"Kitty", miau};
    struct animal pato = {"Lucas", cua};

    hacer_sonido(&perro);
    hacer_sonido(&gato);
    hacer_sonido(&pato);
}
```

El programa mostrado arriba imprime: Fido hace guau guau, Kitty hace miau miau, y Lucas hace cua cua –en ese orden. La invocación de diferentes funcionalidades bajo una misma llamada, `(pa->sonido) ()`, es posible gracias a que cada variable (`perro`, `gato`, y `pato`), a pesar de ser del mismo tipo (`struct animal`), fue asociada con una función diferente. Esto ocurrió en el `main`, al momento de inicializar las variables citadas. Así se hace parecer que `pa->sonido` tiene distintas implementaciones que dependen de cada instancia de `struct animal`.

PROBLEMAS PROPUESTOS

1. Implemente la función `Cruce`, que recibe dos funciones continuas, `f` y `g`, un intervalo `[a, b]`, y retorna un punto `x` en `[a, b]` donde se cumple: $f(x) = g(x)$. Asuma que sólo existe un punto que satisface tal propiedad.
2. Implemente la función `Derivada`, que retorna la derivada de `f` evaluada en el punto `x`. `f` y `x` son pasados como parámetros de `Derivada`.
3. Implemente la función `Aplica`, que recibe una función, `f`, y un arreglo de números reales, `arr`. La función actualiza cada elemento `arr[i]` con el valor `f(arr[i])`.
4. Utilice la función $F(f, n, x) = f(f(f(\dots f(x) \dots)))$ para verificar la siguiente aproximación:

$$\sqrt{2} = 1 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \ddots}}}}.$$

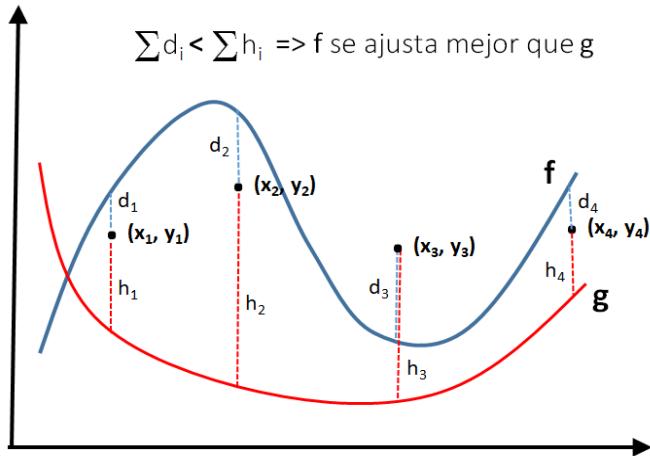
- 5 . Defina punteros a `foo1`, `foo2` y `foo3`, cuyos prototipos se muestran abajo. No use `void*`.

```
void foo1(int arr[], int N);
int* foo2(void);
int (*foo3(void))[5];
```

- 6 . Complete los recuadros para asignar `fs` y `&fs` a los tipos adecuados. No use `void*`.

```
float f1(int x) { return 1.0f/x; }
float f2(int x) { return 5.0f/x; }
int main(void) {
    float (*fs[2])(int) = {f1, f2};
      = fs;
      = &fs;
}
```

7. Implemente la función MejorAjuste, que recibe: un arreglo de dos funciones, f y g , y una serie de N puntos (x_i, y_i) . Se debe retornar la función (f ó g) que se ajuste mejor a los puntos. Por ejemplo, para los 4 puntos mostrados abajo, MejorAjuste tendría retornar la función f .



8. Determine la salida generada por el siguiente código.

```
int main(void) {
    int (*p1)(int *);
    int (*p2[3])(int [4]);
    int (*p3[3][4])(int *);
    void (*p4)(int [3][4]);

    printf("%lu\n", sizeof(p1));
    printf("%lu\n", sizeof(p2));
    printf("%lu\n", sizeof(p3));
    printf("%lu\n", sizeof(p4));
}
```

CAPÍTULO 10

ASIGNACIÓN DINÁMICA DE MEMORIA

Existen situaciones donde es imposible saber de antemano cuánto espacio de memoria van a requerir nuestros programas. Si uno quisiera calcular números primos durante un minuto o simplemente almacenar los divisores de un número arbitrario N , por ejemplo, no sería posible predecir cuántos números primos o divisores serán encontrados y, por ende, no podríamos determinar con exactitud el espacio de memoria que sería necesario para almacenar estos datos. En este tipo de situaciones el uso de variables plantea un serio inconveniente. Guardar los números primos o los divisores en arreglos nos obligaría a tener que «tantear» un tamaño para nuestro arreglo. Si declaramos un arreglo muy pequeño, terminaremos perdiendo datos; si declaramos uno muy grande, estaremos malgastando espacio en memoria.

La forma ideal de lidiar con este tipo de situaciones consiste en hacer que nuestros programas manipulen manualmente la memoria, para que puedan solicitar más espacio justo en el momento en que se detecta la necesidad de almacenar nuevos datos; es decir, durante la ejecución del programa. El proceso de pedir y liberar memoria manualmente durante la ejecución de un programa se denomina **asignación dinámica de memoria**. Cuando usamos variables (arreglos), en cambio, el espacio de memoria reservado para nuestros datos queda fijo desde la declaración de estas variables (arreglos), y ya no puede ser extendido en tiempo de ejecución.

Utilizando asignación dinámica de memoria podremos guardar datos sin necesidad de definir variables. La idea central detrás de este nuevo paradigma consiste en solicitar **bloques de memoria** de tamaño arbitrario para colocar allí nuestros datos manualmente, cuidando de que unos no se sobrepongan a otros. Mientras que una variable entera siempre ocupará 4 bytes y un arreglo de 100 elementos siempre contendrá 100 elementos, los bloques de memoria de los que estamos hablando pueden extenderse o reducirse de tamaño para poder acumular más o menos datos en distintos momentos del programa, según se necesite. Estos bloques de memoria siempre nos serán asignados dentro del segmento de memoria denominado montículo (*heap*, en inglés).

SOLICITAR UN BLOQUE DE MEMORIA

Para mostrar cómo se gestiona manualmente la memoria comenzaré por la sencilla tarea de colocar tres enteros, 10, 20 y 30, dentro del montículo. Primero debemos notar que cada entero ocupa 4 bytes en memoria, ya sea que se guarde en el montículo, en la pila o en cualquier otro segmento. Para lograr nuestro objetivo de grabar tres enteros debemos entonces empezar solicitando un bloque de 12 bytes. Esto se logra usando la siguiente sintaxis:

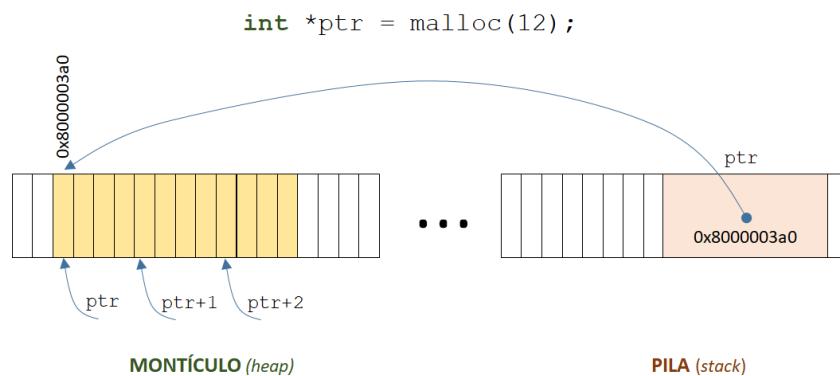
```
int* ptr = malloc(12);
```

La función `malloc`, cuyo prototipo se encuentra en el archivo `stdlib.h`, se encarga de solicitar al sistema un bloque de 12 bytes consecutivos. `malloc` retorna la dirección del bloque reservado, la cual es almacenada en el puntero `ptr`. Para ser precisos, cuando hablamos de la dirección de un bloque nos referimos a la dirección de su primer byte.

El puntero `ptr` será de vital importancia en nuestro programa. Gracias a este podremos: (1) acceder al bloque reservado para grabar allí manualmente nuestros datos, y (2) liberar el bloque cuando ya no necesitemos los datos que allí se almacenan. `ptr` también ocupará espacio en memoria: 8 bytes en la pila, por tratarse de una variable automática (local no estática).

Vale decir que no es posible utilizar la función `sizeof` para obtener el tamaño de un bloque asignado por `malloc`. Es responsabilidad del programador mantener este dato a buen recaudo.

La distribución de la memoria luego de la asignación mostrada podría quedar de la siguiente forma:



Como se ve en el gráfico, los 12 bytes que ahora tenemos a nuestra disposición dentro del montículo pueden ser accedidos desde un puntero (de 8 bytes) almacenado en la pila.

Aún falta el segundo y definitivo paso: grabar los enteros 10, 20 y 30 dentro del bloque reservado.

GRABANDO EN EL MONTÍCULO

Para grabar datos en el montículo se debe indicar explícitamente la dirección donde se guardará cada uno. En nuestro caso, para almacenar tres datos enteros sin que se sobrepongan el primero deberá ocupar las cuatro primeras celdas del bloque; el segundo, desde la 5^{ta} hasta la 8^{va} celda; y el tercero, desde la 9^{ta} hasta la 12^{ta} celda.

Afortunadamente el puntero `ptr` apunta a la primera celda del bloque. Esto implica que `ptr+1` y `ptr+2` apuntarán 4 y 8 bytes a la derecha de `ptr`, justamente a la 5^{ta} y 9^{ta} celda, desde donde conviene empezar a guardar el segundo y el tercer dato, respectivamente. El código para grabar nuestros tres enteros en un bloque de 12 bytes en el montículo sería así:

```
*ptr = 10;
*(ptr+1) = 20;
*(ptr+2) = 30;
```

Hablando con mayor detalle, debe decirse que la función `malloc` retorna un puntero genérico, `void*`; es decir, una dirección sin tipo. En el ejemplo en discusión esta dirección fue capturada en un puntero de tipo `int*`. Esto es totalmente factible puesto que, como ya vimos antes, un puntero genérico puede ser asignado a cualquier tipo de puntero. La razón por la que usamos un puntero de tipo `int*` en el ejemplo que estamos discutiendo es que ya sabíamos de antemano que nuestro objetivo era grabar datos enteros. Al definir el puntero `ptr` de tipo `int*` pudimos utilizar expresiones sencillas, como `ptr+1, ptr+2, ...`, para obtener las direcciones donde debíamos grabar nuestros datos de forma sucesiva y de modo que no se traslapen.

LIBERACIÓN DE MEMORIA

Supongamos que ya hemos concluido con todos los cálculos que teníamos que realizar con nuestros datos. En este caso convendría liberar los 12 bytes de memoria que estamos utilizando.

La liberación de memoria no significa que el sistema «borra» todos los bits del bloque liberado, como algunos suelen creer. Los datos ya grabados bien podrían permanecer incólumes cierto tiempo después de la liberación. Liberar un bloque de memoria simplemente significa que dicho bloque, o parte de este, podrá ser reasignado en posteriores llamadas de `malloc`. Por lo tanto, el programador ya no debería grabar ni leer datos de un bloque liberado, pues otras partes del programa podrían estar sobrescribiendo sobre esa misma zona de memoria.

Para liberar un bloque de memoria se utiliza la función `free`, con prototipo en `stdlib.h`:

```
free(ptr);
```

VERIFICANDO LA DISPONIBILIDAD DE MEMORIA

La función `malloc` no siempre logrará reservar el espacio de memoria solicitado, pues bien podría ocurrir que no haya tanto espacio libre como el que uno solicita. Si este fuera el caso, la función `malloc` retornaría `NULL`. Por ello, una buena práctica de programación consiste en verificar la nulidad del puntero returnedo por `malloc` antes de empezar a utilizarlo. Esto puede implementarse como sigue:

```
int main(void) {
    int* ptr = malloc(3*sizeof(int));
    if(ptr != NULL) {
        *ptr = 10;
        *(ptr+1) = 20;
        *(ptr+2) = 30;
        free(ptr);
    } else
        puts("No hay suficiente espacio en memoria");
}
```

FUGA DE MEMORIA

Es un grave error perder la dirección de un bloque de memoria reservado con el comando `malloc`. Sin esta dirección no sólo será imposible liberar dicha zona de memoria posteriormente, sino que tampoco podremos acceder a ella. Habríamos reservado una zona de memoria que no podríamos usar ni dejar que otros usen. Un caso típico de fuga de memoria se da con el siguiente patrón:

```
int *ptr;
ptr = malloc(1000); // primer bloque
ptr = malloc(5000); // segundo bloque
```

La reutilización de `ptr` ha hecho que se pierda la dirección del primer bloque de 1000 bytes, el cual ya no podrá ser utilizado ni liberado. Si añadiéramos la instrucción `free(ptr)` al final de estas líneas, sólo lograríamos liberar el segundo bloque de memoria, de 5000 bytes. Intentar luego una segunda llamada a `free(ptr)` con la esperanza de liberar el primer bloque tampoco serviría de ayuda. Liberar una zona de memoria que ya ha sido previamente liberada conduce a un comportamiento inesperado del programa.

GRABANDO DATOS DE DIFERENTES TIPOS

Con el siguiente ejemplo veremos cómo se puede grabar datos de distintos tipos en un mismo bloque de memoria. En concreto, grabaremos los datos 10, 'A' y 3.1416 de tipos `int`, `char` y `float`, respectivamente. Podríamos empezar así:

```
void* ptr = malloc(sizeof(int)+ sizeof(char)+ sizeof(float));
```

Con la instrucción anterior se ha solicitado un bloque de 9 bytes en el montículo. Este espacio es justo y suficiente para colocar los datos mencionados.

Note que ahora `ptr` ha sido definido como puntero genérico, `void*`. Esto es una diferencia con respecto al ejemplo anterior. Ahora `ptr+1` y `ptr+2` ya no se refieren a las direcciones donde deberíamos guardar el siguiente y el siguiente dato. De la aritmética de punteros genéricos sabemos que `ptr+N` se refiere a la dirección de la celda que se encuentra N bytes a la derecha de `ptr`. Así pues, nuestros tres datos deberían ser grabados en `ptr`, `ptr+4` y `ptr+5`, dejando cuatro bytes para el primer dato, uno para el segundo, y usando los restantes para el tercero.

Lamentablemente, conocer las direcciones donde debemos grabar cada dato no es suficiente. No basta con especificar una sentencia como `* (ptr+4) = 'A'` y esperar que la letra '`A`' se grabe en el quinto byte del bloque reservado. A pesar de que la sentencia tiene perfecto sentido nos reencontraríamos con un error de programación conocido: No se puede desreferenciar punteros genéricos. Debemos pues indicarle al sistema que lo que se va a grabar en `ptr+4` es un dato de tipo `char`. Esto se logra con una transformación de tipos, haciendo `(char*) (ptr+4)`. Recién ahora que `ptr+4` ya tiene un tipo, podemos grabar en esta dirección haciendo:

```
* ((char*) (ptr+4)) = 'A'
```

En analogía con lo anterior, el código necesario para grabar nuestros datos sería el siguiente:

```
* ((int*)ptr) = 10;
* ((char*) (ptr+4)) = 'A';
* ((float*) (ptr+5)) = 3.1416;
```

GRABANDO ESTRUCTURAS DE DATOS

Así como es posible almacenar datos atómicos en el montículo, también es posible guardar datos agregados. Para ilustrar el asunto guardaremos 100 puntos (`a`, `b`) aleatorios, digamos que con `a` y `b` en $[0, 1]$, dentro del montículo.

Para esto primero debemos definir una estructura que admita datos de la forma (`a`, `b`), lo cual puede hacerse con el siguiente código:

```
typedef struct Tupla {
    double x;
    double y;
} Punto;
```

Luego debemos solicitar el espacio que se necesita para almacenar 100 elementos del tipo definido anteriormente, lo que se logra haciendo:

```
Punto *ptr = malloc(100 * sizeof(Punto));
```

También pudo usarse el nombre del tipo, `struct Tupla`, en lugar del alias, `Punto`.

Como todos los elementos que vamos a guardar son del tipo `Punto`, la expresión `ptr+i` hará referencia a la dirección donde deberá grabarse el $(i+1)$ -ésimo `Punto`.

Las direcciones `ptr+i` son punteros a estructuras y, por lo tanto, podemos usar el operador flecha, `->`, para acceder directamente a los atributos de la estructura apuntada. Para almacenar la $(i+1)$ -ésima estructura en el bloque que nos ha sido asignado bastaría con escribir:

```
(ptr + i) ->x = 1.0f * rand() / RAND_MAX;  
(ptr + i) ->y = 1.0f * rand() / RAND_MAX;
```

Las formulas del lado derecho de las asignaciones simplemente sirven para generar valores aleatorios en $[0,1]$. En un programa completo la variable `i` debería variar desde 0 hasta 99. Se deja al lector que termine el programa que acabamos de bosquejar.

SOLICITANDO MÁS MEMORIA

Como ya se dijo al inicio del capítulo, los bloques de memoria reservados por medio de la función `malloc` pueden cambiar de tamaño en tiempo de ejecución. El comando utilizado para realizar esta solicitud es el siguiente:

```
realloc(ptr, new_size)
```

El primer parámetro, `ptr`, es la dirección del bloque cuyo tamaño se desea modificar; el segundo, `new_size`, es un entero que especifica el nuevo tamaño deseado (en bytes). La función retorna un puntero genérico, `void*`, que contiene la dirección del bloque que acaba de ser extendido o acortado.

Note que muchas veces no será posible, por cuestiones de espacio, extender un bloque que se encuentra apretado entre otros dos bloques previamente reservados. En general, la ampliación de un bloque de memoria a veces podría requerir una reubicación del mismo, i.e. una reorganización de la memoria (*reallocation*). La nueva dirección del bloque luego de su reubicación es retornada por `realloc`. Cuando el sistema reubica un bloque toda la data contenida en el mismo es automáticamente copiada a la nueva dirección.

Para ilustrar mejor lo que acaba de decirse deconstruyamos el siguiente fragmento de código:

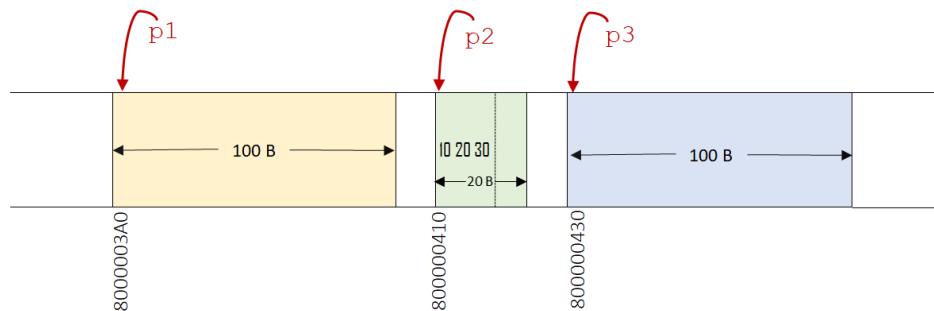
```
int* p1 = malloc(100);
int* p2 = malloc(20);
int* p3 = malloc(100);

*p2 = 10;
*(p2+1) = 20;
*(p2+3) = 30;

int *ptr = realloc(p2, 100);
```

En las tres primeras líneas se han reservado tres bloques de memoria (dentro del montículo). El primer y tercer bloque ocupa cada uno un espacio 100 bytes; el segundo ocupa apenas 20 bytes. Dentro de este bloque central, se han registrado tres datos: 10, 20 y 30 (líneas 4-6).

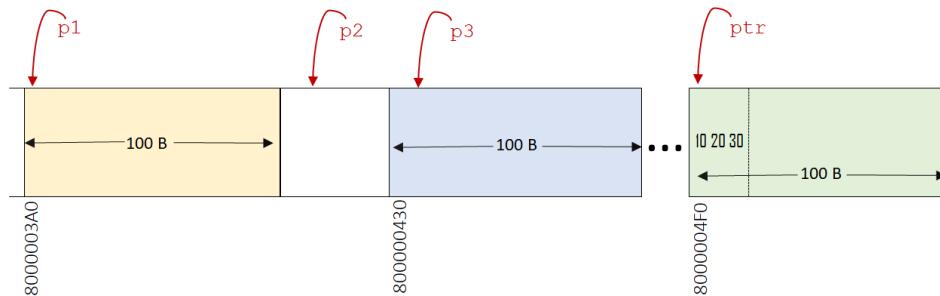
En mi sistema, el código anterior conduce a la siguiente configuración del montículo:



Viendo el gráfico anterior, se nota que el segundo bloque no podrá extenderse a un tamaño de 100 bytes, a menos que sea trasladado a otro lugar de la memoria. Esto es justamente lo que ocurre al hacer:

```
int *ptr = realloc(p2, 100);
```

La línea anterior hace que el segundo bloque, junto con todos sus datos, sea extendido y reubicado en una nueva dirección, que quedará especificada en `ptr`. Depurando mi programa, pude observar que el montículo cambió a la siguiente configuración:



Como el segundo bloque no tenía espacio suficiente para crecer hasta 100 bytes en su posición original, tuvo que ser trasladado a una zona con mayor espacio libre (pero siempre dentro del montículo). Si se quisiera añadir un cuarto dato a este bloque, se podría hacer:

```
* (ptr+3) = 40;
```

`realloc` también libera el espacio de memoria que el bloque ocupaba originalmente. Esto significa que una nueva solicitud de memoria, p.ej. `malloc(10)`, podría ahora reservar algunos de los bytes que originalmente fueron ocupados por el segundo bloque.

El puntero `p2` ha quedado en el aire. Una sentencia como `free(p2)` no solamente sería innecesaria sino que ocasionaría un comportamiento inesperado del sistema.

También pudo hacerse `p2 = realloc(p2, 100)` para reutilizar el puntero `p2` y no tener que declarar un nuevo puntero `ptr`.

OTRAS FORMAS DE SOLICITAR MEMORIA

Digamos algunas cosas de la función `calloc`, cuyo prototipo tiene la siguiente forma:

```
void* calloc(size_t nitems, size_t size)
```

Esta función también se utiliza para reservar un bloque de memoria en el montículo. Pero en lugar de especificar el tamaño del bloque en número de bytes, tal como se hace con `malloc`, `calloc` requiere que se le pase el número de elementos, `nitems`, de tamaño `size` que uno espera almacenar. Otra diferencia (más valiosa) es que la función `calloc` «limpia» el bloque que acabamos de reservar. El bloque reservado estará inicialmente lleno de ceros en cada uno de sus bits. En cambio, cuando se usa `malloc`, no se puede asumir que el espacio reservado estará lleno de ceros, pues podría contener varios bits encendidos.

PROBLEMAS RESUELTOS

1. Implementar una función `BuscaDivisores` que encuentre y grabe los divisores de `N` en un bloque en el montículo. La función deberá luego retornar la dirección de dicho bloque de memoria a la función llamadora para que, desde allí, se puedan imprimir los divisores encontrados en `BuscaDivisores`. El entero `N` es un parámetro de la función.

Solución:

Dado que no se puede conocer de antemano cuantos divisores tendrá `N`, tampoco se puede saber cuantos bytes serán necesarios para acomodarlos en la memoria.

Lo que haremos será empezar pidiendo 4 bytes para almacenar la unidad, pues, por lo pronto, de lo único que podemos estar seguros es que el número 1 será un divisor de `N`.

```
int* ptr = malloc(sizeof(int));
*ptr=1;
```

Luego buscaremos otros divisores de `N`. Sabemos que `num` es divisor de `N` cuando lo divide exactamente, $N \% num == 0$. Usaremos un bucle para hacer que `num` varíe desde 2 hasta `N`.

```
for(int num=2; num<=N; num++)
    if(N % num == 0)
        // num es un divisor de N
```

Cada vez que encontramos un divisor solicitaremos 4 bytes adicionales de memoria para almacenar allí al nuevo divisor encontrado. Recuerde que para invocar a la función `realloc` se necesita pasarle el nuevo tamaño del bloque extendido, no la cantidad de bytes adicionales que necesitamos. Por ello, es conveniente conocer en todo momento la cantidad de divisores que vamos encontrando.

El código para extender el bloque y almacenar el nuevo divisor encontrado es así:

```
int cont=1;
for(int num=2; num<=N; num++)
    if(N % num == 0) {
        cont++;
        ptr = realloc(ptr, cont*sizeof(int));
        *(ptr+cont-1)= num;
    }
```

La variable `cont` se actualiza cada vez que se encuentra un nuevo divisor. En dicho momento, el espacio necesario para almacenar `cont` divisores sería `cont*sizeof(int)`. Note que `cont` se inicializa en 1 porque antes del bucle ya tenemos almacenado un divisor dentro del bloque, la unidad.

El último divisor encontrado debe guardarse siempre en `ptr+cont-1` porque esa es la dirección de los últimos 4 bytes del bloque que inicia en `ptr`. El código completo sería así:

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

int* BuscaDivisores(int N, int *ptam) {
    int *ptr = malloc(sizeof(int));
    *ptr = 1;

    int cont = 1;
    for(int num=2; num<=N; num++)
        if(N % num == 0) {
            cont++;
            ptr = realloc(ptr, cont*sizeof(int));
            *(ptr+cont-1) = num;
        }
    *ptam = cont;
    return ptr;
}

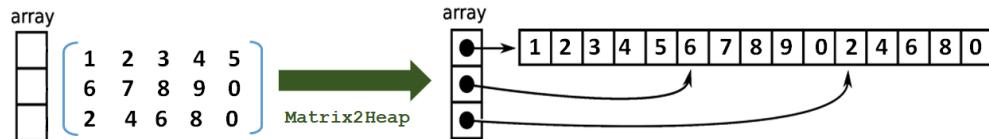
int main(void) {
    int tam;
    int *p= BuscaDivisores(24, &tam);
    for(int i=0; i<tam; i++)
        printf("%d ", *(p+i)); // imprime 1,2,3,4,6,8,12,24
    free(p);
}
```

`BuscaDivisores` recibe un entero `N` y retorna un puntero al bloque de memoria donde se acaban de grabar los divisores de `N`. Con este puntero los divisores pueden ser leídos desde `main`. Pero para ello se necesita un dato más. La función llamadora necesitará, además de la dirección del bloque, el número de divisores que contiene el mismo. Este último dato se obtiene de la variable `tam`.

Desde el `main`, la variable `tam` se pasa por referencia a la función `BuscaDivisores`. Dentro de esta, `tam` es actualizada con el valor de `cont`, en `*ptam = cont`.

La funcionalidad que acabamos de implementar no se podría replicar usando arreglos. Aparte de que el tamaño de los arreglos es fijo, trabajar con estos acarrearía un segundo problema: Si se hubiera almacenado los divisores de `N` en un arreglo definido dentro de `BuscaDivisores`, no se habrían podido leer sus datos desde el `main`. El hipotético arreglo sería una variable local, almacenada en la pila, y su espacio de memoria sería liberado justo antes de retornar a `main`.

2. Implemente una función `Matrix2Heap` que reciba una matriz `M` y un arreglo `array`. Los elementos de la matriz deben ser transferidos al montículo. Tal como se observa en el gráfico, todos los elementos deben guardarse en un mismo bloque; las direcciones de inicio de cada fila serán almacenadas en `array`. El arreglo `array` está en la pila.



Solución:

Definiremos una función `Matrix2Heap` que reciba como parámetros: una matriz `M` de dimensiones arbitrarias `nfils × ncols`, y un arreglo `array` con `nfils` elementos. Dado que `array` debe guardar direcciones de memoria, deberá ser declarado como arreglo de punteros.

La cabecera de `Matrix2Heap` podría quedar de la siguiente forma:

```
void Matrix2Heap( int nfils, int ncols,
                  int M[nfils][ncols], int *array[nfils] )
```

Su primera tarea es pedir un bloque que pueda almacenar `nfils × ncols` elementos.

```
int *ptr = malloc(nfils * ncols * sizeof(int));
```

Luego resuelve el problema en dos pasos. Primero asigna el arreglo `array` con las direcciones que le corresponden. Como se ve en el gráfico, estas están espaciadas una de otra una distancia equivalente a la que ocupan `ncols` elementos. O sea, el arreglo `array` debe almacenar las direcciones `ptr`, `ptr+ncols`, `ptr+2*ncols`, etc. En un segundo paso se almacenan los elementos de `M` en el bloque apuntado por `ptr`, con `*ptr++ = M[i][j]`.

El código de la función pedida se muestra a continuación:

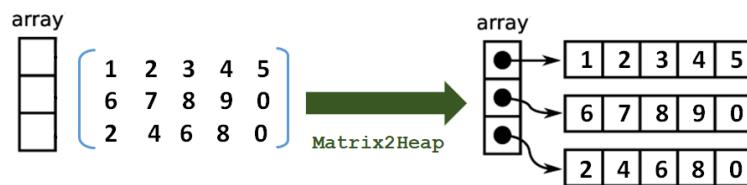
```
void Matrix2Heap2(int nfils, int ncols,
                  int M[nfils][ncols],
                  int* array[nfils])
{
    int *ptr = malloc(nfils * ncols * sizeof(int));

    for(int i=0; i<nfils; i++)
        array[i] = ptr + i*ncols;

    for(int i=0; i<nfils; i++)
        for(int j=0; j<ncols; j++)
            *ptr++ = M[i][j];
}
```

Para grabar los elementos de M consecutivamente dentro del bloque reservado el programa actualiza el valor de `ptr` en una unidad para cada nuevo dato, i.e. `*ptr++`. Aquí no hay riesgo de fuga de memoria; el valor de `ptr` es sobreescrito muchas veces; pero la dirección de inicio del bloque sigue estando disponible en `array[0]` y, por lo tanto, este siempre podrá ser liberado.

3. Implemente una función `Matrix2Heap` que reciba una matriz M y un arreglo `array`. Los elementos de la matriz deben ser transferidos al montículo. Tal como se observa en el gráfico, cada fila de M debe guardarse como un bloque independiente dentro del montículo; las direcciones de los bloques serán almacenadas en `array`. El arreglo `array` está en la pila.



Solución:

Para resolver este problema definiremos como parámetros de la función `Matrix2Heap` una matriz M de dimensiones arbitrarias `nfilas x ncols`, y un arreglo `array` de `nfilas` elementos, el cual deberá declararse como arreglo de punteros. La cabecera de `Matrix2Heap` podría ser así:

```
void Matrix2Heap( int nfilas, int ncols,
                  int M[nfilas][ncols], int *array[nfilas] )
```

La transferencia de los elementos de M al montículo nos exige tener que navegar la matriz. Durante este proceso, apenas se visite la i -ésima fila de M , se puede aprovechar para solicitar un bloque de tamaño `ncols*sizeof(int)` en el montículo y guardar su dirección en `array[i]`. Luego, durante el recorrido de esta fila, cuando se visite su j -ésimo elemento, $M[i][j]$, se puede aprovechar para guardar éste en la j -ésima posición del i -ésimo bloque, o sea en `array[i]+j`.

El siguiente programa implementa la función pedida:

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

void Matrix2Heap(int nfilas, int ncols,
                 int M[nfilas][ncols],
                 int *array[nfilas])
{
    for(int i=0; i<nfilas; i++) {
        array[i] = malloc(ncols * sizeof(int));
        for(int j=0; j<ncols; j++)
            *(array[i]+j) = M[i][j];
    }
}
```

```

int main(void) {
    int *array[3];
    int M[3][5] = {{1, 2, 3, 4, 5},
                    {6, 7, 8, 9, 0},
                    {2, 4, 6, 8, 0}};
    Matrix2Heap(3, 5, M, array);
    for(int i=0; i<3; i++)
        for(int j=0; j<5; j++)
            printf("%d ", *(array[i]+j));
}

```

La función `main` imprime los elementos de la matriz `M` desde el montículo.

4. Implemente una función `ubicaNum` que guarde los pares (i, j) tales que $M[i][j]=\text{number}$. Las posiciones (i, j) deben ser almacenadas en el montículo y deben ser accesibles desde la función llamadora. `M` y `number` son pasados como parámetros.

Solución:

Guardaremos la secuencia de índices (i, j) como una lista de enteros. Cada vez que encontramos una nueva instancia de `number` en la posición (i, j) de la matriz `M`, la lista deberá crecer en 8 bytes para poder almacenar dos enteros más, i y j .

La función podría quedar así:

```

int* ubicaNum(int fils, int cols, int M[fils][cols], int number, int* p){
    int cont = 0;
    int *ptr = NULL;

    for(int i=0; i<fils; i++)
        for(int j=0; j<cols; j++)
            if(M[i][j] == number) {
                cont += 2;

                if(cont==2)
                    ptr = malloc(cont*sizeof(int));
                else
                    ptr = realloc(ptr, cont*sizeof(int));

                *(ptr+cont-2) = i;
                *(ptr+cont-1) = j;
            }
    *p = cont;
    return ptr;
}

```

Se usa un doble bucle para recorrer los elementos de la matriz `M`. Cada vez que se cumple $M[i][j]=\text{number}$, se graba (i, j) en el montículo, lo que puede realizarse de dos maneras:

- Si se cumple `cont==2`, significa que recién se grabará el primer par (i, j) en el montículo y, por ello, se solicita espacio para esos dos enteros, `malloc(cont*sizeof(int))`.
- Si no se cumple `cont==2`, significa que ya hay varios datos en el montículo. En ese caso se debe extender el espacio de un bloque ya existente. Como la variable `cont` se incrementa en dos unidades, `cont += 2`, apenas se encuentra `number` en `M`, la expresión `cont*sizeof(int)` representa el tamaño que debería tener el bloque luego de su extensión. La solicitud para extender el bloque actual se implementa con `realloc(ptr, cont*sizeof(int))`.

En ambos casos, ya sea usando `malloc` para solicitar un bloque o `realloc` para extenderlo, se deben grabar los valores (i, j) . Esto se hace en las dos últimas líneas del bloque `for`, con las sentencias: $*(\text{ptr}+\text{cont}-2)=i$ y $*(\text{ptr}+\text{cont}-1)=j$. Estas instrucciones almacenan i y j en posiciones adyacentes.

Las dos últimas líneas de `ubicaNum` ya no pertenecen al bucle. La instrucción `*p = cont` sirve para sacar el valor `cont` fuera de `ubicaNum` a través de una variable pasada por referencia. La sentencia `return ptr` retorna la dirección del bloque grabado a la función llamadora.

Una forma de evaluar la función `ubicaNum` sería la siguiente:

```
int main(void) {
    int nrep;
    int M[3][5] = {{2, 7, 5}, {3, 5, 7}, {7, 7, 5}};
    int *ptr = ubicaNum(3, 5, M, 7, &nrep);

    if(ptr != NULL)
        for(int i=0; i<nrep; i+=2)
            printf("\n(%d, %d)", *(ptr+i), *(ptr+i+1));
}
```

El programa mostrado imprime: $(0, 1)$, $(1, 2)$, $(2, 0)$ y $(2, 1)$, pues esas son las posiciones donde se encuentra `number=7` en la matriz $M_{3 \times 5}$.

Dichos valores son obtenidos a partir del puntero `ptr` que fue retornado desde `ubicaNum`. El número de elementos almacenados a partir de `ptr` es `nrep`. Esta variable es pasada por referencia a `ubicaNum` para que su valor sea asignado dentro de esta función.

PROBLEMAS PROPUESTOS

1. Defina un arreglo de números (`short`) y almacene sus direcciones dentro del montículo.
2. Almacene la dirección de la función `float inversa(int)` en el montículo. Luego invoque a la función desreferenciando su dirección almacenada en el montículo.
3. Definir una función `GrabaFibonacci` que almacene los números de la serie de Fibonacci menores que `N` en el montículo. La función debe retornar una dirección para que los elementos almacenados puedan ser leídos desde la función llamadora. `N` es un parámetro de la función.
4. El siguiente programa imprime cinco direcciones de memoria. Diga a qué segmentos de memoria (código, pila, etc.) corresponde cada dirección.

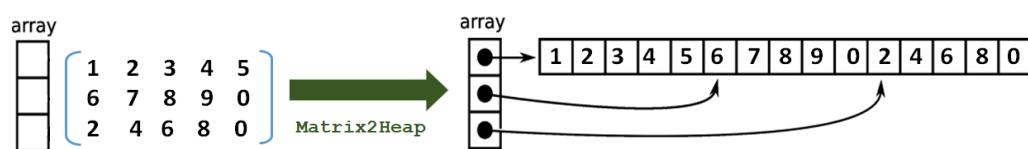
```
#include <stdio.h>
#include <stdlib.h>

typedef struct alumno {
    char *nombre;
    double nota;
} Alumno;

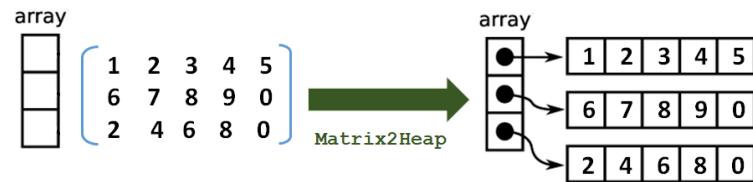
int main(void)
{
    Alumno *p = malloc(sizeof(Alumno));
    p->nombre = "Pedro";
    p->nota = 9.7;

    printf("%p\n", p);
    printf("%p\n", &p);
    printf("%p\n", p->nombre);
    printf("%p\n", &(p->nombre));
    printf("%p\n", malloc);
}
```

5. Definir una función `QuitaRepetidos` que reciba un arreglo y almacene sus elementos (sin incluir repetidos), dentro del montículo.
6. Implemente una función `Matrix2Heap` que reciba una matriz `M` y un arreglo `array`. Tal como se ve en el gráfico, los elementos deben guardarse en un mismo bloque; las direcciones de inicio de cada fila serán almacenadas en `array`. El arreglo `array` debe estar dentro del montículo.



7. Implemente una función `Matrix2Heap` que reciba una matriz `M` y un arreglo `array`. Tal como se observa en el gráfico, cada fila de la matriz debe guardarse como un bloque independiente dentro del montículo; las direcciones de los bloques serán almacenadas en `array`. El arreglo `array` debe estar dentro del montículo.



CAPÍTULO

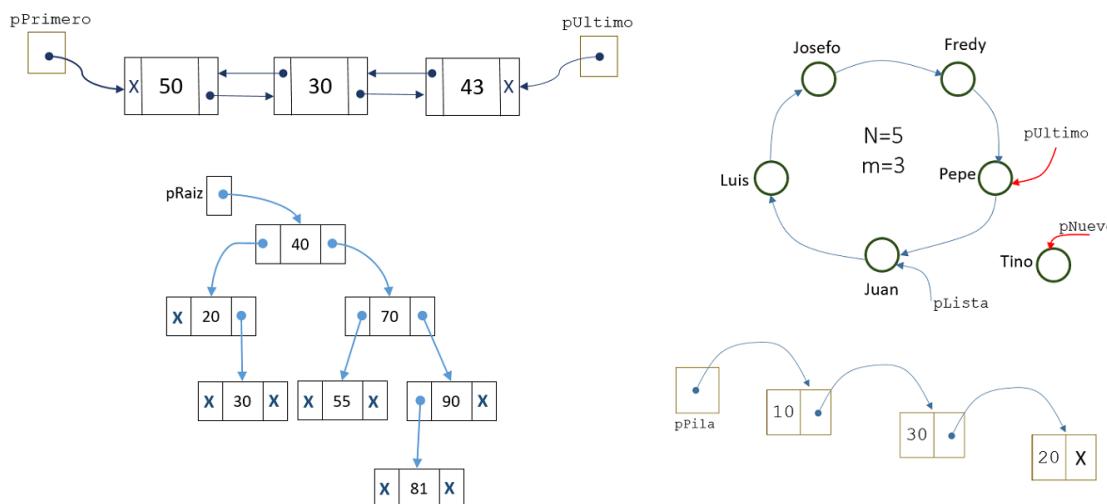
1 1

ESTRUCTURAS DINÁMICAS DE DATOS

Una estructura dinámica de datos es un conjunto de nodos enlazados a través de punteros. Los nodos son creados para ocupar espacio dentro del montículo (*heap*), lo que nos permite extender y/o reducir el tamaño de estas estructuras durante la ejecución del programa, creando y/o eliminando nodos dinámicamente, según las necesidades.

Los nodos se implementan como estructuras (*struct*). Estas pueden contener varios atributos, dependiendo de la información que se necesite almacenar; pero por lo menos uno de estos atributos debe ser un puntero al mismo tipo de estructura. Concretamente, un nodo de tipo *struct miNodo* debe tener al menos un atributo de tipo *struct miNodo**. De esta forma es posible crear grupos de nodos que se apunten entre sí. Las estructuras que poseen atributos que apuntan al mismo tipo de estructura se denominan **estructuras de datos recursivas**.

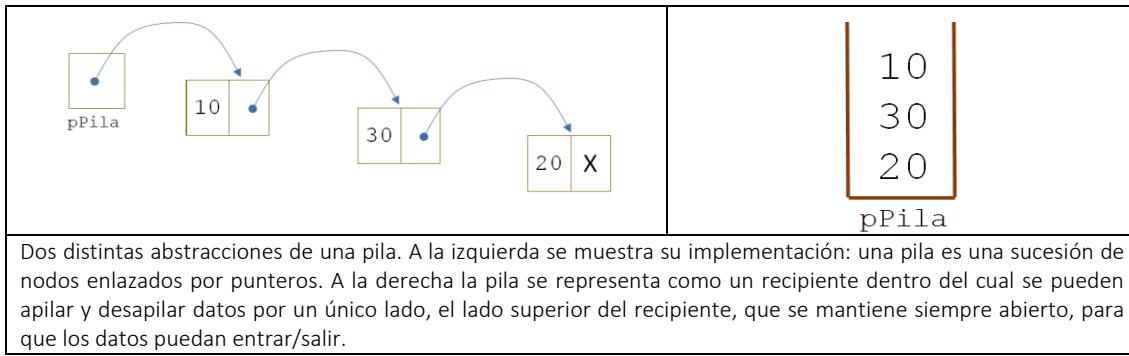
Dependiendo de la manera como se enlacen sus nodos, las estructuras dinámicas pueden tener distintas topologías. Algunas de éstas se muestran abajo:



Según su topología, existen estructuras lineales y no lineales. Las primeras se caracterizan porque cada uno de sus nodos está conectado, como mucho, a otros dos, que pueden considerarse su antecesor y su sucesor. Las pilas, colas y listas son algunos ejemplos de este tipo de estructuras. Por otro lado, las estructuras no lineales tienen conexiones más intrincadas. Los árboles en todas sus variantes (binarios, B, rojinegros, etc.) son ejemplos de estructuras no lineales.

PILA

Una pila es una secuencia de nodos. Lo que diferencia a esta estructura dinámica de datos de otras estructuras lineales es la forma en que se añaden y eliminan sus elementos. Cada vez que se añade o se retira un nodo de la pila se sigue la regla LIFO: El último nodo en ingresar será el primero en salir (*Last In, First Out*). El siguiente gráfico muestra una pila con tres nodos.



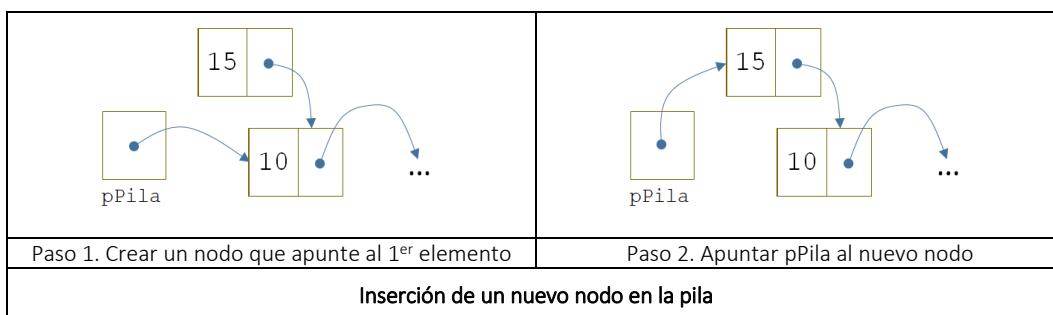
En la pila mostrada en el gráfico anterior cada nodo posee dos atributos: un valor entero y un puntero al siguiente nodo. En cuanto al último nodo de la pila, su puntero tiene el valor `NULL`, que en el gráfico se representa por el símbolo `X`. También que se necesita un puntero `pPila` para apuntar hacia el primer nodo de la pila, desde donde podremos acceder a todos los demás. El puntero `pPila` no forma parte de ninguna estructura.

Para implementar los nodos de la pila mostrada podemos hacer:

```
typedef struct nodo {  
    int valor;  
    struct nodo *pSig;  
} Nodo;  
  
Nodo *pPila = NULL;
```

INSERCIÓN DE UN NODO EN UNA PILA

La inserción de un nuevo nodo se hará con la función `push`. Esta debe recibir como argumento el valor del nodo a insertar. El siguiente gráfico muestra las alteraciones sufridas por la pila `10->30->20` luego de invocar `push(15)`.



Primero se debe crear el nuevo nodo a insertar. Su atributo `valor` debe ser 15 o, en general, el argumento usado en la invocación de `push`. Su atributo `pSig` debe contener la dirección del primer nodo de la pila. Esto último se debe a que la regla LIFO exige que el nodo a insertar debe ir siempre al inicio de la pila.

Posteriormente, como segundo paso, se debe redireccionar el puntero `pPila` para que apunte al nuevo nodo, el que se acaba de crear. De este modo el nuevo nodo pasaría a ser el primer elemento. En términos programáticos, esto se implementa de la siguiente manera:

```
void push(int dato) {
    Nodo *pNuevo = malloc(sizeof(Nodo));
    pNuevo->valor = dato;
    pNuevo->pSig = pPila;
    pPila = pNuevo;
}
```

Procedimiento para apilar elementos

Las tres primeras líneas de la función `push` implementan lo que en el diagrama anterior se denominó Paso 1. Inicialmente, el nuevo nodo es creado dentro del montículo, con la función `malloc`, en la dirección almacenada en `pNuevo`. Luego se hace `pNuevo->valor = dato` para que el valor del nuevo nodo sea el `dato` pasado como parámetro a `push`. Note que es posible utilizar el operador flecha porque `pNuevo` es una dirección, su valor fue retornado desde la función `malloc`. El atributo `pNuevo->pSig` se asigna con la dirección del primer nodo, que está contenida en `pPila`.

El segundo paso, Paso 2, se implementa en la última línea de `push`. Se redirecciona `pPila` para que apunte al nodo recién creado; es decir, `pPila=pNuevo`. Con esto el nuevo nodo se convierte en el primer nodo de la pila: Es apuntado desde `pPila` y apunta a la cadena de nodos restantes.

EXTRACCIÓN DE UN NODO DE UNA PILA

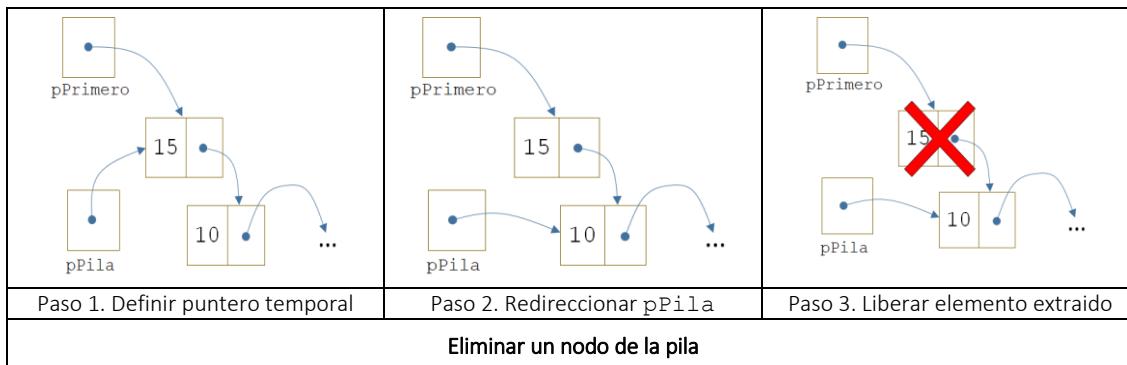
Para eliminar elementos de la pila se utilizará la función `pop`. Esta no necesita recibir parámetros. Se sobreentiende que en cada invocación, `pop()`, se retirará el último elemento que fue insertado; en nuestro caso, 15. Esto en cumplimiento de la regla LIFO. Más bien, la función `pop` suele retornar un dato, el valor del elemento extraído.

Para extraer un elemento de la pila se procede de modo inverso a la inserción. Primero se redirecciona `pPila` para que apunte al segundo elemento, de modo que el primero quede ignorado, fuera de la pila. Finalmente, se debe liberar el elemento que acaba de ser retirado para que no siga ocupando espacio en memoria. Esto se implementa de la siguiente forma:

```
int pop(void) {
    Nodo *pPrimero = pPila;
    int valor_ext = pPrimero->valor;
    pPila = pPrimero->pSig;
    free(pPrimero);
    return valor_ext;
}
```

Procedimiento para desapilar elementos

Note que se necesita declarar un puntero adicional, `pPrimero`, para que guarde la dirección del primer nodo, i.e. el valor almacenado en `pPila`. Luego de ejecutar la primera línea de la función `pop`, la pila quedaría como se muestra abajo, en Paso 1.



Segundo, se debe redireccionar `pPila` para que apunte al segundo nodo, pues el primero está pronto a desaparecer. Note que si `pPila` apunta al primer nodo, entonces la dirección del segundo nodo puede conseguirse con `pPila->pSig` (o también con `pPrimero->pSig`). Luego de modificar el valor de `pPila`, la estructura quedaría como se muestra en el Paso 2.

Finalmente, se debe liberar el espacio que ocupaba el que hasta hace poco fue el nodo inicial. Justamente para eso se necesitó definir `pPrimero`. La liberación del nodo extraído se hace con `free(pPrimero)`. No se podría hacer `free(pPila)` porque `pPila` ya está apuntando a un nodo distinto del que se desea eliminar.

Abajo se muestra el código necesario para implementar la pila mostrada anteriormente.

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

typedef struct nodo {
    int valor;
    struct nodo *pSig;
} Nodo;

Nodo *pPila = NULL;

void push(int dato) {
    Nodo *pNuevo = malloc(sizeof(Nodo));
    pNuevo->valor = dato;
    pNuevo->pSig = pPila;
    pPila = pNuevo;
}

int pop(void) {
    Nodo *pPrimero = pPila;
    int valor_ext = pPrimero->valor;
    pPila = pPila->pSig;
    free(pPrimero);
    return valor_ext;
}
```

```

void show(void) {
    Nodo *pTmp = pPila;
    while(pTmp != NULL) {
        printf("%d --> ", pTmp->valor);
        pTmp = pTmp->pSig;
    }
}
int main(void) {
    push(20); push(30); push(10); push(15);
    show();
}

```

El programa anterior incluye la función `show`, que permite visualizar los elementos de la pila empezando desde el primer nodo (el más cercano a `pPila`) hasta el último (el que apunta a `NULL`). Dicha función utiliza un puntero temporal, `pTmp`, que empieza apuntado al primer nodo (`pTmp=pPila`), imprime el dato que este contiene (`pTmp->dato`), pasa a apuntar al siguiente nodo (`pTmp=pTmp->Sig`), y así sucesivamente, hasta llegar al final de la pila, `pTmp==NULL`.

Para comprender por qué `pTmp==NULL` es una condición de parada debe notarse que el último nodo de la pila siempre tiene el atributo `pSig` apuntando a `NULL`. La primera vez que se invocó a `push` se ejecutó `pNuevo->pSig = pPila` en un momento en que la pila aún estaba vacía (`pPila=NULL`).

Uno también podría quitar algunos elementos de la pila e imprimir el valor que contienen haciendo algo como `var = pop();` para luego imprimir el entero `var`. Es conveniente verificar que la pila contiene elementos (`pPila!=NULL`) antes de invocar a `pop`.

UNA PILA GENERALIZADA

El programa anterior es una versión simple pero limitada de cómo implementar una pila. El problema es que todas las funciones (`push`, `pop` y `show`) asumen la presencia de una variable global `pPila`. Esto no es grave si deseamos trabajar con una única pila; pero sí queremos utilizar varias dentro del mismo programa, éste debe ser generalizado para evitar la redundancia de código. En una versión más generalizada, las funciones `push`, `pop` y `show` deberán recibir un puntero a la pila que se desea modificar y/o imprimir.

Es fácil subestimar el problema descrito y pensar que puede subsanarse con sólo introducir la variable `pPila` dentro del `main` y pasársela como argumento a `push`, `pop` y `show`, con algo como:

```

int main(void) {
    Nodo *pPila = NULL;
    push(20, pPila); push(30, pPila); push(10, pPila); push(15, pPila);
    show(pPila);
}

```

Pero lo anterior es incorrecto (incluso si alteramos las definiciones de `push`, `pop` y `show`). La razón radica en que `push` y `pop` necesitan cambiar el valor de `pPila`. Cada vez que apilamos o desapilamos, `pPila` deja de apuntar a un elemento para apuntar a otro nuevo. La modificación de `pPila` dentro de una

función exige que `pPila` sea pasada por referencia, no por valor. Sería un error hacer una invocación como `push(20,pPila)`. Esto no es paso por referencia. Para que el valor de `pPila` pueda ser cambiado dentro de `push` se debe pasar una referencia a esta variable. Lo correcto es invocar `push(20,&pPila)`.

Dado que `pPila` es del tipo `Node*`, `&pPila` sería del tipo `Node**`.

A diferencia de `push` y `pop`, la función `show` es la única que sí podría recibir `pPila` por valor. Esto es debido a que esta función sólo necesita leer el valor de `pPila`, pero no cambiarlo.

El programa generalizado quedaría como se muestra a continuación:

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

typedef struct nodo {
    int valor;
    struct nodo *pSig;
} Nodo;

void push(int dato, Nodo** ptr_pPila) {
    Nodo *pNuevo = malloc(sizeof(Nodo));
    pNuevo->valor = dato;
    pNuevo->pSig = *ptr_pPila;
    *ptr_pPila = pNuevo;
}

int pop(Nodo** ptr_pPila) {
    Nodo *pPrimero = *ptr_pPila;
    int valor_ext = pPrimero->valor;
    *ptr_pPila = (*ptr_pPila)->pSig;
    free(pPrimero);
    return valor_ext;
}

void show(Nodo* pPila) {
    Nodo *pTmp = pPila;
    while(pTmp != NULL) {
        printf("%d --> ", pTmp->valor);
        pTmp = pTmp->pSig;
    }
}

int main(void) {
    Nodo *pPila = NULL;

    push(20,&pPila); push(30,&pPila); push(10,&pPila); push(15,&pPila);
    show(pPila);

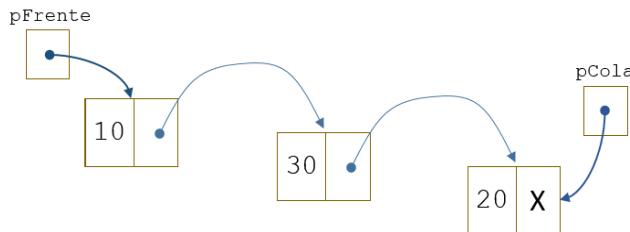
    int valor = pop(&pPila);
    printf("Se extrajo %d", valor);
}
```

El código de las funciones `push` y `pop` parece muy diferente; pero la lógica sigue siendo la misma. Lo que antes funcionó correctamente con `pPila` ahora funcionará con `*ptr_pPila`, donde el parámetro `ptr_pPila` es asignado con `&pPila` en la invocación de `push` y `pop`, dentro del `main`.

COLA

Una cola es una secuencia de nodos cuyos elementos se añaden y/o eliminan siguiendo la regla FIFO: El primer elemento en ingresar será el primero en salir (*First In, First Out*), tal como ocurre en las colas de los supermercados, por ejemplo.

Como se observa en el siguiente gráfico, una de las posibles implementaciones de una cola utiliza dos punteros: `pFrente` apunta al primer elemento de la cola, el que está a punto de ser atendido; y `pCola` apunta al último, a partir del cual se colocan los nuevos elementos que ingresan a la cola.



Al igual que con las pilas, los nodos de una cola se implementan como estructuras recursivas. Inicialmente `pFrente` y `pCola` apuntan a `NULL`.

```

typedef struct nodo {
    int valor;
    struct nodo *pSig;
} Nodo;

Nodo *pFrente = NULL;
Nodo *pCola = NULL;
  
```

INSERCIÓN DE UN NODO EN UNA COLA

La inserción de un nodo en una cola se realiza invocando a la función `enqueue`, que podría implementarse como se muestra abajo:

```

void enqueue(int dato) {
    Nodo *pNuevo = malloc(sizeof(Nodo));
    pNuevo->valor = dato;
    pNuevo->pSig = NULL;

    if(pFrente == NULL) {
        pFrente = pNuevo;
        pCola = pNuevo;
    } else {
        pCola->pSig = pNuevo;
        pCola = pNuevo;
    }
}
  
```

En las tres primeras líneas se construye el nuevo nodo que ingresará a la cola. Su atributo `valor` contendrá el dato pasado como parámetro a la función `enqueue`.

Luego prosigue la inserción. Antes de insertar un nuevo nodo a una cola debe verificarse si ésta está vacía. En dicho caso (`pFrente==NULL`), el nodo insertado será a la vez el primero y el último. Esto se indica haciendo que `pFrente` y `pCola` apunten a `pNuevo`. En el caso más general, cuando la cola ya contiene elementos, el último nodo deberá apuntar a `pNuevo` (`pCola->pSig=pNuevo`), quien entonces pasaría a convertirse en el último elemento de la cola (`pCola=pNuevo`).

EXTRACCIÓN DE UN NODO DE UNA COLA

La eliminación de un elemento de una cola se realiza invocando a la función `dequeue`, que podría implementarse como se muestra abajo:

```
int dequeue(void) {
    int valor_ext = pFrente->valor;
    Nodo *pPrimero = pFrente;
    pFrente = pFrente->pSig;
    free(pPrimero);
    return valor_ext;
}
```

En una cola siempre se extrae el primer nodo, que está apuntado por `pFrente`. El valor de este nodo es guardado en `valor_ext` y su dirección en `pPrimero`. El valor `valor_ext` es returned por la función; y `pPrimero` es usado para liberar la memoria que estaba siendo ocupada por el primer nodo. Luego que el primer nodo es sacado de la cola hay un nuevo primer nodo. Esto se implementa actualizando el valor de `pFrente` con `pFrente = pFrente->pSig`.

Los lectores más sutiles se darán cuenta que `pCola` no está siendo alterada dentro de `dequeue`, ni siquiera en el caso de que la cola quede vacía. Esta omisión no implica dificultad alguna siempre que se utilice `pFrente` y no `pCola` para verificar si la cola está vacía o no.

La función `show` imprime los elementos de la cola y es igual a la que se usó para imprimir pilas.

UNA COLA GENERALIZADA

El código anterior es una versión simplificada de cola. Funciona correctamente si trabajamos con una sola cola; pero si quisieramos utilizar varias en un mismo programa tendríamos escribir una terna de funciones `enqueue`, `dequeue` y `show` para cada cola. Esto debido a que dichas funciones hacen referencia a un par específico de variables globales. En una versión más generalizada, `enqueue`, `dequeue` y `show` deberían recibir los punteros `pFrente` y `pCola` como parámetros, para no tener que buscarlos en el entorno global.

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

typedef struct nodo {
    int valor;
    struct nodo *pSig;
} Nodo;

void enqueue(int dato, Nodo** ptr_pFrente, Nodo** ptr_pCola) {
    Nodo *pNuevo = malloc(sizeof(Nodo));
    pNuevo->valor = dato;
    pNuevo->pSig = NULL;

    if(*ptr_pFrente == NULL) {
        *ptr_pFrente = pNuevo;
        *ptr_pCola = pNuevo;
    } else {
        (*ptr_pCola)->pSig = pNuevo;
        (*ptr_pCola) = pNuevo;
    }
}

int dequeue(Nodo** ptr_pFrente) {
    int valor_ext = (*ptr_pFrente)->valor;
    Nodo *pPrimero = *ptr_pFrente;
    *ptr_pFrente = (*ptr_pFrente)->pSig;
    free(pPrimero);
    return valor_ext;
}

void show(Nodo* pFrente) {
    Nodo *pTmp = pFrente;
    while(pTmp != NULL) {
        printf("%d <- ", pTmp->valor);
        pTmp = pTmp->pSig;
    }
}

int main(void) {
    Nodo *pFrente = NULL;
    Nodo *pCola = NULL;

    enqueue(10, &pFrente, &pCola); enqueue(30, &pFrente, &pCola);
    enqueue(20, &pFrente, &pCola);
    show(pFrente);

    int valor = dequeue(&pFrente);
    printf("El cliente %d fue atendido primero", valor);
}
```

Dentro del `main`, `pFrente` y `pCola` son pasadas por referencia a `enqueue` y `dequeue` (el `&` que precede a sus nombres así lo delata). Esto es necesario porque `pFrente` y `pCola` podrían cambiar de

valor dentro de enqueue y dequeue. Dicha alteración sería imposible usando el paso por valor. Sólo show se basta con el valor de pFrente, sin necesidad de su referencia. Esto porque show nunca intenta reasignarle un nuevo valor a pFrente, sólo lee su valor actual.

ÁRBOL BINARIO

A diferencia de las pilas y las colas, los árboles son estructuras no lineales. En el caso particular del árbol binario, cada uno de sus nodos puede apuntar a 0, 1 o 2 nodos distintos. Los nodos apuntados se denominan nodos hijos; el nodo que apunta, nodo padre. Cada nodo se implementa como una estructura que contiene dos punteros que potencialmente podrían apuntar hacia el hijo derecho y hacia el hijo izquierdo. Estos punteros también pueden tomar el valor NULL para indicar la ausencia de algún hijo.

Por su posición en un árbol los nodos pueden ser de tres tipos: (1) la raíz, es el único nodo que no posee padre, (2) las hojas, son los nodos que no poseen hijos, y (3) los nodos intermedios, son aquellos que no caben en ninguna de las dos definiciones anteriores.

ÁRBOL BINARIO DE BÚSQUEDA

El árbol binario de búsqueda es un tipo especial de árbol binario. Se construye de modo que todos los valores que se encuentran a la izquierda de un nodo sean menores que el valor del nodo en cuestión; y, análogamente, todos los valores a su derecha deben ser mayores.

<pre> graph TD 40 --> 20 40 --> 70 20 --> 30 70 --> 55 70 --> 90 90 --> 81 </pre>	<pre> pRaiz[] --> 40[40] 40 --> 20[X 20 X] 40 --> 70[X 70 X] 20 --> 30[X 30 X] 70 --> 55[X 55 X] 70 --> 90[X 90 X] 90 --> 81[X 81 X] </pre>
Representación esquemática de un árbol binario	Implementación de un árbol binario. El nodo 40 es la raíz del árbol; los nodos 30, 55 y 81 son hojas; el resto son nodos intermedios. Las X's representan un puntero nulo.

Si, por ejemplo, nos fijamos en el nodo 40 del gráfico anterior, vemos que todos los descendientes de su rama derecha (70, 55, 90, 81) poseen valores mayores que 40; y toda la descendencia que proviene de su rama izquierda (20, 30) posee valores menores que 40. Lo mismo se cumple para cualquier otro nodo.

A veces conviene visualizar cada nodo como si fuera la raíz de un árbol que posee un subárbol derecho y otro izquierdo. Así, los nodos 90 y 81, por ejemplo, conforman el subárbol derecho del nodo 70.

El árbol binario permite agilizar la búsqueda de datos. Se puede demostrar matemáticamente que si uno almacena un mismo grupo de datos en un árbol binario y en una estructura de datos lineal (p.ej. lista o arreglo), la búsqueda de un dato cualquiera dentro del árbol es, en promedio, más eficiente.

El siguiente código muestra una forma de implementar un árbol binario de búsqueda.

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

typedef struct nodo {
    int valor;
    struct nodo *pIzq;
    struct nodo *pDer;
} Nodo;

void insertar(int dato, Nodo** ptr_pArbol) {
    // crear nuevo nodo
    Nodo *pNuevo = malloc(sizeof(Nodo));
    pNuevo->valor = dato;
    pNuevo->pIzq = NULL;
    pNuevo->pDer = NULL;

    // insertar nuevo nodo
    if(*ptr_pArbol == NULL)
        *ptr_pArbol = pNuevo;
    else {
        if(dato < (*ptr_pArbol)->valor)
            insertar(dato, &((*ptr_pArbol)->pIzq));
        else
            insertar(dato, &((*ptr_pArbol)->pDer));
    }
}

void mostrar_PreOrder(Nodo *pRaiz) {
    if(pRaiz != NULL) {
        printf("%d ", pRaiz->valor);
        mostrar_PreOrder(pRaiz->pIzq);
        mostrar_PreOrder(pRaiz->pDer);
    }
}

int main(void) {
    Nodo *pRaiz = NULL;

    insertar(40, &pRaiz);
    insertar(20, &pRaiz); insertar(70, &pRaiz);
    insertar(30, &pRaiz); insertar(55, &pRaiz);
    insertar(90, &pRaiz);
    insertar(81, &pRaiz);

    mostrar_PreOrder(pRaiz);
}
```

Cada elemento del árbol se implementa como una estructura del tipo `Nodo`. Esta posee dos punteros, `pIzq` y `pDer`, que enlazan cada nodo con un subárbol izquierdo y otro derecho.

Los nodos son añadidos al árbol a través de la función `insertar`. En términos simples, la regla para insertar un nodo en un árbol binario de búsqueda es la siguiente: insertar un dato en un árbol A consiste en insertar el dato en el subárbol izquierdo de A (si el dato es menor que la raíz de A), o en su subárbol derecho (si el dato es mayor que la raíz de A). Así se continúa recursivamente hasta que, eventualmente, nos tocará insertar el nuevo nodo en un subárbol vacío, i.e. cuya raíz es `NULL`.

Note que sería un error invocar `insertar(40, pRaiz)`, omitiendo el operador de dirección en `&pRaiz`. Dentro de la función `insertar` se tiene modificar el valor de `pRaiz` (p.ej. cuando el árbol está vacío). Para que ello sea posible dicho puntero debe ser pasado por referencia, no por valor.

La impresión del árbol se realiza con `mostrar_PreOrder`. Esta también es una función recursiva, como muchas de las que se utilizan para manipular árboles. Imprimir un árbol consiste en imprimir el valor de su raíz para luego imprimir su subárbol izquierdo y después su subárbol derecho. Dicho esto, el árbol mostrado en el gráfico anterior se imprime como 40 20 30 70 55 90 81. Esta no es la única forma de navegar e imprimir un árbol. Ademas del recorrido preorden, existen también el recorrido inorder y postorder.

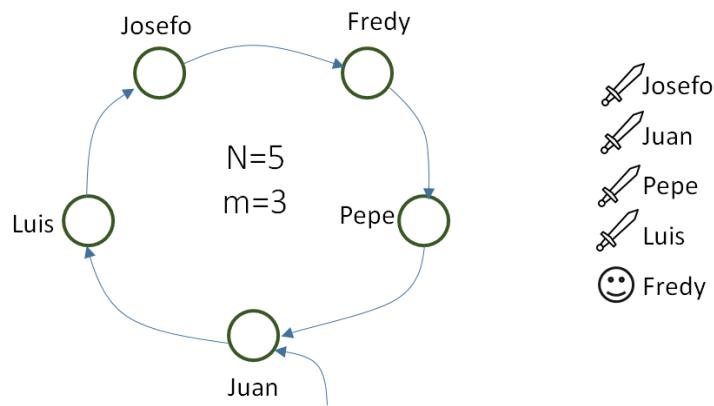
PROBLEMAS RESUELTOS

1. Listas circulares. Problema de Josefo

En el primer siglo de nuestra era, un grupo de 40 judíos cuyo escondite había sido cercado por unos soldados romanos decidieron quitarse la vida para evitar ser capturados y condenados a una humillante esclavitud. El procedimiento con que acordaron matarse sería el siguiente: Todos se colocarían alrededor de un círculo. Luego, el primero mataría al segundo y pasaría la espada al tercero; éste mataría al cuarto y pasaría la espada al quinto; y así sucesivamente. Al final, el último sobreviviente tendría que quitarse la vida por mano propia.

La intuición matemática del historiador Flavio Josefo, uno de los cuarenta desgraciados, le permitió anticipar la posición del último que habría de inmolarse y así pudo colocarse en dicho lugar. La razón por la que esta historia ha llegado hasta nuestros días es porque Josefo acertó con su predicción, tal como narró en su libro «*La guerra de los judíos*» (aprox. 75 D.C.). Pero entonces decidió romper su acuerdo previo con los que ya eran cadáveres, no se quitó la vida, se dejó capturar, y se entregó a una esclavitud que momentos antes fingió despreciar.

Probablemente ni el propio Josefo imaginó que esa deshonrosa conducta suya y la traición *post mortem* a sus 39 correligionarios daría lugar a un interesante problema matemático, que ha tenido innumerables variantes y que, en su momento, llamó la atención de genios de la talla de Euler.



En forma generalizada, el problema de Josefo consiste en determinar la posición del sobreviviente de un círculo conformado por N individuos, que serán eliminados uno por uno, cada m personas.

Se le pide implementar una solución al problema de Josefo para $N=5$ y $m=3$. Como se muestra en el gráfico anterior, se debe mostrar el orden en que serán eliminados los $N-1$ individuos e identificar al sobreviviente. Asuma que el recorrido de la lista circular se hace en sentido horario.

Solución:

Tal como sugiere el gráfico, el círculo de muerte puede ser implementado como una lista circular. Cada nodo apunta al siguiente; y el último, al primero. Además se necesita guardar el nombre de cada individuo para poder distinguirlos y anunciar sus decesos conforme vayan ocurriendo.

Por lo tanto, el nodo que utilizaremos como elemento constitutivo de nuestra lista será similar a los que ya definimos para otros tipos de estructuras; sólo se diferenciará en el tipo del primer atributo, que esta vez será `char*`. En concreto, los nodos se crearán a partir del siguiente tipo:

```
typedef struct nodo {
    char *nombre;
    struct nodo *pSig;
} Nodo;
```

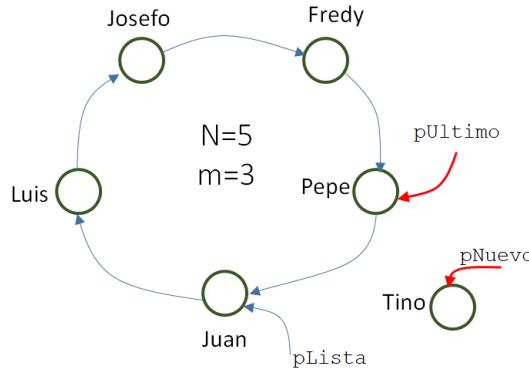
En cuanto a la inserción, cada nodo nuevo será añadido al final del círculo, después del último nodo que haya sido insertado. El código para añadir `nombre` a una lista apuntada por `*ptr_pLista` es:

```
void add(Nodo** ptr_pLista, char* nombre) {
    Nodo *pNuevo = malloc(sizeof(Nodo));
    Nodo *pUltimo = ultimoNodo(*ptr_pLista);
    pNuevo->nombre = nombre;
    if(pUltimo != NULL) {
        pUltimo->pSig = pNuevo;
        pNuevo->pSig = *ptr_pLista;
    } else {
        pNuevo->pSig = pNuevo;
        *ptr_pLista = pNuevo;
    }
}
```

El puntero `pNuevo` contiene la dirección del nodo que va a introducirse a la lista; `pUltimo` apunta al último nodo de la lista. (Ver gráfico en la siguiente página).

Cuando la lista no está vacía (`pUltimo!=NULL`), el nuevo nodo debe pasar a ocupar la última posición de la lista, `pUltimo->pSig=pNuevo`. Por tratarse de una lista circular, el nuevo último nodo debe apuntar al nodo inicial, `pNuevo->pSig = *ptr_pLista`.

En el caso especial cuando la lista está vacía (`pUltimo==NULL`), el nodo a insertar sería el único elemento y, en tal situación, debe apuntarse a sí mismo, `pNuevo->pSig=pNuevo`. Adicionalmente, la dirección de la lista circular debe ser actualizada para apuntar a `pNuevo`; esto se logra con: `*ptr_pLista=pNuevo`.



Nodo "Tino" antes de ser insertado en la lista circular

La función `ultimoNodo`, invocada desde `add`, se implementa de la siguiente forma:

```
Nodo* ultimoNodo(Nodo* pLista) {
    Nodo* pTmp = NULL;
    if(pLista != NULL) {
        pTmp = pLista;
        while(pTmp->pSig != pLista)
            pTmp = pTmp->pSig;
    }
    return pTmp;
}
```

Esta función retorna `NULL` cuando la lista aún está vacía (`pLista==NULL`). De lo contrario, se utiliza un puntero `pTmp` para recorrer la lista, elemento por elemento, hasta llegar al último nodo, aquel que apunta al nodo inicial, `pTmp->pSig==pLista`.

Finalmente, la eliminación serial cada `m` individuos puede implementarse en la función `main`:

```
int main(void) {
    Nodo *pLista = NULL;
    int m=3;
    int N=5;

    anadeElementos(&pLista, N);

    Nodo *pTmp = pLista;
    Nodo *pAnt = ultimoNodo(pLista);

    do {
        for(int i=1; i<m; i++) {
            pTmp = pTmp->pSig;
            pAnt = pAnt->pSig;
        }

        printf("\nSe murió %s", pTmp->nombre);
        pAnt->pSig = pTmp->pSig;
        free(pTmp);
        pTmp = pAnt->pSig;
    } while (pTmp != pAnt);

    printf("\nSobrevivió %s", pTmp->nombre);
}
```

La función `anadeElementos` invoca a la función `add` previamente estudiada. Aquí se añaden N nodos a la lista `pLista`. Los nombres de los nodos a añadir podrían salir de cualquier fuente.

Luego viene la temible caminata de m pasos. Esta se implementa como un bucle `for`. Al final del bucle, un puntero `pTmp` terminará apuntando al nodo que se va a eliminar; otro puntero, `pAnt`, terminará apuntando al nodo anterior a `pTmp`.

La eliminación se inicia con el anuncio del nodo que será eliminado, con `printf`. Luego de la mala nueva, el nodo `pTmp` se retira (lógicamente) de la lista. Para esto, el nodo que le antecedia pasa a conectarse directamente con su sucesor, `pAnt->pSig = pTmp->pSig`. Recién en ese momento el nodo `pTmp` es físicamente eliminado con `free(pTmp)`. Luego la caminata vuelve a iniciarse desde el nodo que sigue a `pAnt`, `pTmp = pAnt->pSig`.

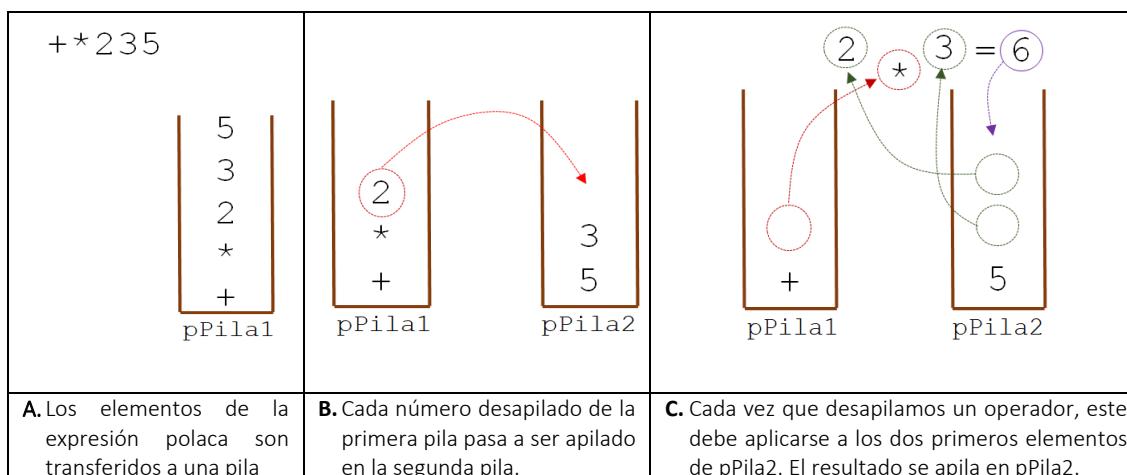
La carnicería culmina cuando queda una sola persona. Dicho momento se reconoce porque es cuando aparece un nodo que se apunta a sí mismo, `pTmp==pAnt`. El sobreviviente sería `pTmp->nombre`.

2. Notación polaca

La notación polaca nos permite escribir cualquier secuencia de operaciones aritméticas sin necesidad de utilizar paréntesis, sin que ello produzca ambigüedades en el orden de evaluación. Se caracteriza porque coloca los operadores a la izquierda de sus operandos. Por ejemplo, las expresiones $2 * 3 + 5$ y $2 * (3 + 5)$ se representarían como: $+ * 235$ y $* 2 + 35$, respectivamente. Esta notación es fácil de computar cuando todas las operaciones involucradas tienen la misma aridad (p.ej. binaria, en el caso de las operaciones aritméticas básicas). Se le pide escribir un programa que permita calcular el resultado de una serie de operaciones representadas en notación polaca.

Solución:

La evaluación de una expresión en notación polaca requiere dos pilas, `pPila1` y `pPila2`. Inicialmente, cada elemento de la expresión debe apilarse en `pPila1` (figura A). Luego, los elementos de `pPila1` serán desapilados uno por uno. Si el elemento desapilado es un número, entonces se apilará en la segunda pila, `pPila2` (figura B). Pero si es un operador, se desapilarán dos elementos de `pPila2`, se les aplicará el operador, y el resultado irá a `pPila2` (figura C).



El proceso culmina cuando la primera pila queda vacía; el resultado se obtiene de la segunda pila que, en ese momento, si la expresión polaca era correcta, sólo tendrá un elemento.

En términos programáticos, los valores que se almacenarán en la primera y segunda pila son distintos. La primera contiene caracteres mientras que la segunda sólo contiene números. Por ello debemos definir dos tipos de estructuras, `NodoChar` y `NodoInt`, tal como se aprecia abajo:

```
typedef struct nodo1 {
    char valor;
    struct nodo1 *pSig;
} NodoChar;

typedef struct nodo2 {
    int valor;
    struct nodo2 *pSig;
} NodoInt;
```

Adicionalmente, para la primera pila, la que almacena caracteres, implementaremos las funciones `pushChar` y `popChar`, que son análogas a las funciones `push` y `pop` que estudiamos anteriormente, pero con la diferencia de que éstas trabajan con nodos de tipo `NodoChar`. Similarmente, la segunda pila, la que almacena números enteros, será manipulada con funciones `pushInt` y `popInt`.

La función que transfiere la expresión polaca a la primera pila es tan simple como:

```
void introduce(char expr[], NodoChar **ptr_pPila) {
    int i=0;
    while(expr[i] != '\0')
        pushChar(expr[i++], ptr_pPila);
}
```

El arreglo `expr` contiene expresiones como `/*235`; sus elementos `expr[i]` son pasados a la primera pila, la que contiene caracteres. Por simplicidad se está asumiendo que los operandos de la expresión polaca sólo poseen un dígito.

La función que evalúa la expresión polaca se muestra en la siguiente página.

Cada elemento que es desapilado de la primera pila es capturado en la variable `simbolo`, desde donde se determina si es un dígito o un operador. Para ello se utiliza la función `isdigit`, cuyo prototipo está en `ctype.h`.

Si el dato contenido en `simbolo` es un dígito, entonces se apila `simbolo - '0'` en la segunda pila. La extraña resta `simbolo - '0'` nos permite obtener el valor numérico contenido en `simbolo` a partir de su código ASCII (p.ej. el número 5 en lugar del código ASCII de '5').

Cuando el dato contenido en `simbolo` es un operador, se desapilan dos elementos de la segunda pila, `op1` y `op2`. En ese momento la segunda pila se queda con dos elementos menos, pero inmediatamente ganará uno nuevo, cuando se le apile el resultado de: `op1 simbolo op2`. Esto se realiza dentro de la sentencia `switch` que, como se observa, sólo considera la suma y el producto como operadores. El lector podría agregar más operadores binarios si así lo desea.

```

int evaluaPolaca(NodoChar **ptr_pPila1, NodoInt **ptr_pPila2) {
    char simbolo;
    while(*ptr_pPila1 != NULL) {
        simbolo = popChar(ptr_pPila1);
        if(isdigit(simbolo)) {
            pushInt(simbolo-'0', ptr_pPila2);
        } else {
            int op1 = popInt(ptr_pPila2);
            int op2 = popInt(ptr_pPila2);
            switch(simbolo) {
                case '+': pushInt(op1+op2, ptr_pPila2); break;
                case '*': pushInt(op1*op2, ptr_pPila2); break;
            }
        }
    }
    return popInt(ptr_pPila2);
}

```

La evaluación termina cuando la primera pila queda vacía luego de tantas desapilaciones, i.e. cuando deja cumplirse la condición `*ptr_pPila1 != NULL`. En dicho momento el resultado se encuentra en la cabeza de la segunda pila y se retorna con `return popInt(ptr_pPila2)`.

El programa principal podría tener la siguiente forma:

```

int main(void) {
    NodoChar *pPila1 = NULL;
    NodoInt *pPila2 = NULL;
    char expr[20] = "+*235";
    introduce(expr, &pPila1);
    int result = evaluaPolaca(&pPila1, &pPila2);
    printf("%s = %d", expr, result);
}

```

Se omitirá las definiciones de las funciones `pushChar`, `popChar`, `pushInt` y `popInt`, por ser estas muy similares a las ya estudiadas `push` y `pop`.

3. Listas enlazadas. Insertar en cualquier posición.

Las pilas y las colas son tipos especiales de listas. Son restrictivas en el sentido que las posiciones de los elementos a insertar y/o eliminar están predeterminadas por las reglas LIFO/FIFO. En el caso general de una lista enlazada, uno puede insertar un nuevo elemento en cualquier posición de la lista y retirar uno existente de cualquier otra. Se le pide implementar el procedimiento de inserción de una lista enlazada.

Solución:

La estrategia será la siguiente: Cuando se pida insertar un nodo en la N -ésima posición de una lista, conseguiremos la dirección del $N-1$ -ésimo nodo. Luego, manipulando punteros, lograremos ejecutar la inserción pedida. Suena simple pero hay que considerar algunos casos especiales. Primero, se

podría querer insertar en la N -ésima posición de una lista que tiene menos de $N-1$ elementos. Segundo, debe considerarse que cuando se tenga que insertar en la primera posición de la lista ($N=1$), no existirá ningún nodo previo, es decir, no tendremos un $N-1$ -ésimo nodo.

Para poder solventar fácilmente el problema de las inserciones infactibles (en posiciones fuera de rango) se va a definir dos variables por cada lista: un puntero al nodo inicial y el número de elementos de la lista. Estas dos variables estarán encapsuladas dentro de una estructura `Lista`, tal como se muestra abajo:

```
typedef struct nodo {
    int valor;
    struct nodo *pSig;
} Nodo;

typedef struct lista {
    Nodo *pInicio;
    int NUM_ELEMS;
} Lista;
```

El atributo `NUM_ELEMS` de `Lista` se usa para abortar inserciones en posiciones fuera de rango.

La función de inserción, `add`, se muestra abajo. Recibe tres parámetros: un puntero `plst` a una `Lista`, el dato a insertar y la posición `pos` de inserción. No retorna valores.

Se considera por separado el caso especial de inserción al inicio de una lista (`pos==1`). Para esto se requieren pasos similares a los ya utilizados al insertar en una pila. En concreto, el nuevo nodo debe cargarse con el dato pasado como parámetro a `add`, y se debe actualizar el puntero al inicio de la lista para que apunte al nuevo nodo, `plst->pInicio = pNuevo`.

```
void add(Lista* plst, int dato, int pos) {
    Nodo *pNuevo = malloc(sizeof(Nodo));
    pNuevo->valor = dato;

    if(pos == 1) {
        pNuevo->pSig = plst->pInicio;
        plst->pInicio = pNuevo;
    } else {
        if(pos-1 <= plst->NUM_ELEMS) {
            Nodo *pTmp = obtenerNodo(plst, pos-1);
            pNuevo->pSig = pTmp->pSig;
            pTmp->pSig = pNuevo;
        } else {
            printf("No se puede insertar en la posicion %d", pos);
            exit(1);
        }
        (plst->NUM_ELEMS)++;
    }
}
```

En el caso general, la inserción en la posición `pos` exige obtener la dirección del nodo que ocupa la posición anterior, `pos-1`. Esto se logra haciendo:

```
Nodo *pTmp = obtenerNodo(plst, pos-1);
```

Luego que se ejecute la línea anterior, `pTmp` quedará apuntando al (`pos-1`)-ésimo nodo. La inserción del nuevo nodo se consigue en dos pasos: (a) haciendo que éste apunte al nodo que actualmente está en `pos` (el sucesor del nodo en `pTmp`). Esto se logra con:

```
pNuevo->pSig = pTmp->pSig;
```

y luego (b) haciendo que el nodo `pTmp` se coloque antes del nodo nuevo, es decir:

```
pTmp->pSig = pNuevo;
```

Ya que la inserción de un dato en `pos` requiere la recuperación del (`pos-1`)-ésimo nodo, la función `add` se asegura que dicho nodo exista, i.e. `pos-1 <= plst->NUM_ELEMS`. La insatisfacción de esta condición indicaría que se está intentando insertar fuera de la lista. Dicho problema es advertido con un mensaje de error, tras lo cual el programa termina, `exit(1)`.

La declaración inicial de una lista puede hacerse con:

```
Lista miLista = {NULL, 0};
```

Esto indica que `miLista` aún está vacía: su nodo inicial es `NULL` y aún posee 0 elementos.

Las invocaciones a la función `add` pueden hacerse de la siguiente forma:

```
add(&miLista, 10, 1);
```

Lo anterior añadirá un nuevo nodo con el valor de 10 en la primera posición de `miLista`.

El siguiente programa muestra una posible implementación de una lista y de cómo podríamos insertar algunos elementos:

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

typedef struct nodo {
    int valor;
    struct nodo *pSig;
} Nodo;

typedef struct lista {
    Nodo *pInicio;
    int NUM_ELEMS;
} Lista;
```

```
Nodo *obtenerNodo(Lista* plist, int k) {
    Nodo *pTmp = plist->pInicio;
    for(int i=1; i<k; i++)
        pTmp = pTmp->pSig;
    return pTmp;
}

void show(Lista lst) {
    Nodo *pTmp = lst.pInicio;
    while(pTmp != NULL) {
        printf("%d <- ", pTmp->valor);
        pTmp = pTmp->pSig;
    }
    printf("\n");
}

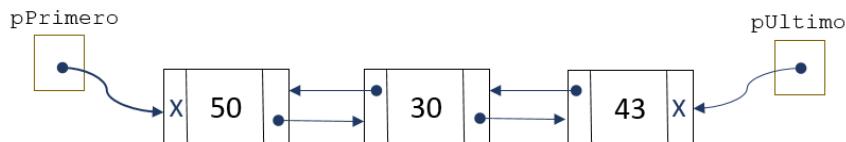
void add(Lista* plist, int dato, int pos) {
    Nodo *pNuevo = malloc(sizeof(Nodo));
    pNuevo->valor = dato;

    if(pos == 1) {
        pNuevo->pSig = plist->pInicio;
        plist->pInicio = pNuevo;
    } else {
        if(pos-1 <= plist->NUM_ELEMS) {
            Nodo *pTmp = obtenerNodo(plist, pos-1);
            pNuevo->pSig = pTmp->pSig;
            pTmp->pSig = pNuevo;
        } else {
            printf("No se puede insertar en la posicion %d", pos);
            exit(1);
        }
    }
    (plist->NUM_ELEMS)++;
}

int main(void) {
    Lista miLista = {NULL, 0};
    add(&miLista, 10, 1); show(miLista);
    add(&miLista, 30, 1); show(miLista);
    add(&miLista, 20, 2); show(miLista);
    add(&miLista, 50, 4); show(miLista);
}
```

PROBLEMAS PROPUESTOS

1. Mencione algunas ventajas/desventajas de usar listas enlazadas en lugar de arreglos.
2. Escriba un programa que determine si los paréntesis de una expresión aritmética están bien balanceados, p.ej. $3*(1+5*(1+1)-3)$, o no, p.ej. $2*((3+6)*7))+1$.
3. En una posición (x,y) de un tablero de $8x8$ se coloca un caballo de ajedrez. Implemente una función que imprima todas las casillas en las que podría ubicarse el caballo exactamente luego de N saltos. El valor N es pasado como parámetro.
4. Implemente la función `addSorted`, que añade el dato pasado como parámetro a una lista enlazada de números enteros. La inserción debe realizarse de tal modo que los elementos de la lista siempre se mantengan ordenados ascendentemente.
5. Implemente la función `removePosition` que elimina el N -ésimo elemento de una lista enlazada. El valor de N es pasado como parámetro.
6. Implemente la función `removeValue` que elimina todos los elementos de una lista enlazada cuyo valor sea igual al dato pasado como parámetro.
7. Implementar una función que resuelva de manera generalizada el problema de Josefo. La función debe recibir: (a) una lista circular, (b) N , el número de nodos, (c) m , el salto, que indica cuántos elementos deben saltarse para eliminar al siguiente, y (d) `dir`, el sentido horario/antihorario en que se visitarán los elementos. La función debe retornar el nombre del sobreviviente.
8. **Lista doblemente enlazada.** En la lista mostrada, cada nodo posee dos punteros, que apuntan al nodo siguiente y al anterior. Se le pide implementar las funciones `add`, `remove` y `show`, que se encarguen de agregar un nuevo nodo en una posición cualquiera, de eliminar un nodo de una posición dada, y de mostrar los elementos de la lista, respectivamente.



9. Existen tres maneras de recorrer los nodos de un árbol binario.
 - a. Recorrido *preOrder*: Recorrer un árbol consiste en visitar su raíz para luego recorrer su subárbol izquierdo y posteriormente el derecho.
 - b. Recorrido *inOrder*: Recorrer un árbol consiste en recorrer su subárbol izquierdo, visitar su raíz y, finalmente, recorrer su subárbol derecho.

- c. Recorrido *postOrder*: Recorrer un árbol consiste en recorrer primero su subárbol derecho, visitar su raíz y, finalmente, recorrer su subárbol izquierdo.

Se le pide implementar tres métodos que permitan visitar e imprimir los nodos de un árbol en los tres posibles órdenes especificados anteriormente.

10. Implemente una función que retorne el número de hojas que posee un árbol.
11. Implemente una función que retorne el número de nodos que posee un árbol.
12. Implemente una función que retorne la altura de un árbol, i.e. el número de niveles que existe desde la raíz hasta la hoja más profunda.
13. Escriba una función que reciba dos árboles binarios de búsqueda y fusione todos sus elementos en un nuevo árbol.