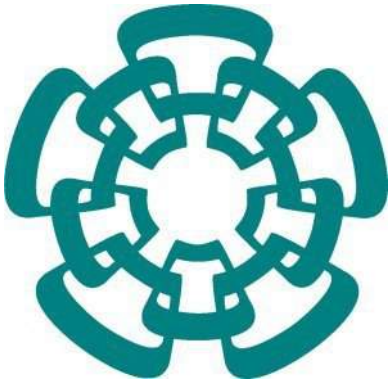


Cinvestav Unidad Tamaulipas

Proceso de Admisión
Maestría en Ciencias en Ingeniería y Tecnologías Computacionales

Programación Básica en Lenguaje C

Dr. Wilfrido Gómez Flores

Índice

1. Introducción

El lenguaje C es un lenguaje de programación de propósito general, ya que no está vinculado a un sólo sistema operativo o tipo de máquina. Este lenguaje fue desarrollado por Dennis Ritchie en los Laboratorios Bell entre 1972 y 1973. De acuerdo con el índice TIOBE, en mayo de 2023, el lenguaje C está posicionado en segundo lugar de los lenguajes de programación más populares[†].

C contiene las construcciones fundamentales de control-flujo requeridos para programas estructurados, por ejemplo, toma de decisiones (`if-else`), seleccionar una opción de un conjunto de casos posibles (`switch`), ciclos con criterio de término (`while`, `for`), entre otras.

Actualmente se suele identificar a C como un lenguaje de *medio nivel*, debido a que interactúa con la capa de abstracción de un sistema informático, ya que sirve como puente entre el hardware y la capa de programación de un sistema informático. Se manipulan objetos como caracteres, números y direcciones de memoria. Estos pueden ser combinados y modificados con los operadores aritméticos y lógicos implementados en las máquinas. C viene acompañado de una biblioteca que contiene funciones para acceder al sistema operativo (e.g., leer y escribir archivos), entrada y salida formateadas, asignación de memoria, manipulación de cadenas, entre otros.

Aunque C se ajusta a las capacidades de muchas computadoras, es independiente de cualquier arquitectura de máquina en particular. Con un poco de cuidado es fácil escribir programas portátiles, es decir, programas que pueden ser ejecutados sin cambios en una variedad de hardware. Además, los compiladores[‡] avisarán de la mayoría de los errores.

1.1. Empezando a programar en C

A continuación se comenzará con una rápida introducción a C. El objetivo es mostrar los elementos esenciales del lenguaje sin entrar en detalles, reglas y excepciones.

[†]https://www.tiobe.com/tiobe-index

[‡]Un compilador es un programa informático que traduce un programa que ha sido escrito en un lenguaje de programación a un lenguaje común, usualmente lenguaje de máquina.

De manera general, se deben seguir los siguientes pasos para hacer funcionar un programa:

1. Escribir el programa.
2. Compilar el programa.
3. Cargarlo y ejecutarlo.

El primer programa para escribir es el mismo para todos los lenguajes de programación: imprimir en pantalla las palabras '`hello, world`'. En el Programa 1 se muestra el código de este primer programa en C.

Programa 1: Primer programa en C.

```
1  // Primer programa en C
2  #include <stdio.h> // Biblioteca de funciones estandar
3
4  int main() // Funcion principal
5  {
6    printf("hello, world\n");
7    return 0;
8  }
9
10 /*
11 Este programa imprime en pantalla una cadena
12 de caracteres con las palabras hello, world
13 */
```

En la línea `2` se incluye la biblioteca de funciones estándar. En la línea `4` se define la función llamada `main`, que no recibe argumentos. Las sentencias dentro de la función `main` se encierran entre llaves. En la línea `6` la función `main` llama a la función `printf` perteneciente a la biblioteca `stdio.h` para imprimir una secuencia de caracteres. La función `printf` nunca suministrará una nueva línea automáticamente, se debe usar `\n` para incluir una nueva línea de caracteres. En las líneas `1` y `10`–`13` se muestran *comentarios* utilizados para explicar el código fuente. Un comentario que comienza con los caracteres `//` no pueden ocupar más de una línea, mientras que los comentarios encerrados entre los caracteres `/*` y `*/` pueden ocupar más de una línea.

Al momento de programar en C (esta es una regla de oro) se debe finalizar cada sentencia con un punto y coma '`;`' (como en la línea `6`), ya que con este

carácter se le indica al compilador que ha finalizado una sentencia. Si se omite el punto y coma habrá errores de sintaxis al momento de compilar el programa. Las directivas como `#include`y `#define`, y la funciones como `main()` no llevan punto y coma debido a que no son sentencias.

La compilación y ejecución de un programa depende del sistema operativo que se esté utilizando. Como un ejemplo específico, sobre el sistema MacOs se debe crear el programa en un archivo cuya extensión sea '`.c`', por ejemplo, '`hello.c`' y luego compilarlo en la línea de comandos del Terminal mediante la instrucción

```
gcc hello.c -o hello2
```

lo cual creará un archivo ejecutable llamado '`hello2`'. Para ejecutar este programa, se escribe el comando

```
./hello2
```

y se imprimirá en pantalla

```
hello, world
```

Ejemplo 1.1: (5 min)

Escribir y ejecutar el programa '`hello, world`' en tu sistema.

2. Elementos básicos de C

Variables y *constantes* son los objetos de datos básicos que se manipulan en un programa. Las *declaraciones* listan las variables que serán usadas, se establecen sus tipos de datos y opcionalmente se definen sus valores iniciales. Los *operadores* especifican cómo se manipularán las variables y constantes. Las *expresiones* combinan variables y constantes para producir nuevos valores. El *tipo* de un objeto determina el conjunto de valores que puede tener y qué operaciones pueden realizarse sobre él. Las *funciones de entrada y salida* (E/S) permiten al programa interactuar con el exterior. A continuación, se estudiarán estos elementos básicos de programación en C.

2.1. Tipos de datos

2.1.1. Básicos

En C existen los siguientes tipos de datos básicos ó fundamentales:

- `char`: un sólo byte capaz de almacenar un carácter.
- `int`: un entero, típicamente refleja el tamaño natural de los enteros en el sistema huésped.
- `float`: punto flotante de precisión simple, no tiene más de siete dígitos significativos.
- `double`: punto flotante de doble precisión, puede tener hasta 16 dígitos significativos.

Los *modificadores* opcionales que pueden acompañar a los tipos básicos son: con signo, sin signo, largo y corto; y se aplican con las palabras clave: `signed`, `unsigned`, `long` y `short`, respectivamente, como se muestra en la Tabla 2.1.

Las funciones `scanf` y `printf` de entrada y salida, respectivamente, soportan una cadena de texto conteniendo códigos y banderas de formato para indicar diferentes tipos y opciones de formato (ver Sección 2.4). En la columna 'Formato' de la Tabla 2.1 se listan los códigos de formato utilizados por cada tipo de datos. Por ejemplo:

```
printf("Color %s, Número %d, Flotante %4.2f", "rojo", 12345, 3.14);
```

Tipo	Modificador		Rango		Formato	Bytes
			Mínimo	Máximo		
`char`		`unsigned`	0	255	`%c`	1
		`signed`	-128	127	`%c`	1
`int`	`short`	`unsigned`	0	$65,535$	`%u`	2
		`signed`	$-32,768$	$32,767$	`%d`	2
	`long`	`unsigned`	0	$4,294,967,295$	`%lu`	4
		`signed`	$-2,147,483,648$	$2,147,483,647$	`%ld`	4
`float`			-3.40×10^{38}	3.40×10^{38}	`%f`	4
`double`			-1.79×10^{308}	1.79×10^{308}	`%lf`	8
	`long`		-1.18×10^{4932}	1.18×10^{4932}	`%Lf`	8

Tabla 2.1: Tipos de datos básicos y sus modificadores.

indica que se imprimirá en pantalla una cadena de caracteres, un número entero, y un número flotante con un ancho de cuatro espacios y dos dígitos decimales.

También existe un tipo `void` que se utiliza para declarar funciones que no retornan un valor o para declarar un puntero a un tipo no especificado. Si `void` aparece entre paréntesis a continuación del nombre de una función, indica que la función no acepta argumentos.

2.1.2. Derivados

Los *tipos derivados* son construidos a partir de los tipos básicos, y algunos de ellos son los siguientes:

1. **Puntero**: es una dirección de memoria que indica dónde se localiza un objeto de un tipo especificado. Para definir una variable de tipo puntero se utiliza el operador de indirección `*` (ver Sección 5).
2. **Estructura**: es una variable que representa lo que normalmente se conoce como registro, esto es, un conjunto de uno o más campos de igual o diferentes tipos (ver Sección 4.3).
3. **Arreglo**: es un conjunto de objetos, todos del mismo tipo, que ocupan posiciones sucesivas en memoria (ver Sección 4.1).
4. **Funciones**: es una subrutina que puede tomar argumentos de varios tipos de datos y retorna un valor de un tipo especificado (ver Sección 6).

2.1.3. Nombres de tipos

Se pueden declarar nuevos nombres de tipos de datos (sinónimos de otros tipos) mediante el especificador `typedef`, ya sean básicos o derivados, los cuales pueden ser utilizados para declarar variables de estos tipos. La sintaxis es la siguiente:

```
typedef tipo_C identificador;
```

donde

- `tipo_C` es cualquier tipo de datos definido en C, básico o derivado.
- `identificador` es el nuevo nombre dado a `tipo_C`.

Por ejemplo, se propone el tipo `ENTERO` como sinónimo de `int` mediante la instrucción

```
typedef int ENTERO;
```

2.2. Variables

Una *variable* representa un espacio de memoria cuyo tipo especifica la cantidad de memoria necesaria para guardar información. El valor de una variable puede cambiar a lo largo de la ejecución de un programa.

Existen ciertas restricciones para definir los nombres de variables. Los nombres están conformados de letras y dígitos; el primer carácter del nombre debe ser siempre una letra. El guión bajo '_' cuenta como una letra; es útil para mejorar la legibilidad de variables con nombres largos, por ejemplo, `valor_temperatura_1`. Letras mayúsculas y minúsculas son diferentes, por ejemplo, `x` y `X` son nombres de variables diferentes. Tradicionalmente las letras minúsculas se usan para variables, mientras que letras mayúsculas se usan para constantes.

Las *palabras clave* son identificadores predefinidos que tienen un significado especial, las cuales son reservadas, es decir, no se pueden usar como nombres de variables. En C se tienen las siguientes palabras clave: `auto`, `break`, `case`, `char`, `const`, `continue`, `default`, `do`, `double`, `else`, `enum`, `extern`, `float`, `for`, `goto`, `if`, `int`, `long`, `register`, `return`, `short`, `signed`, `sizeof`, `static`, `struct`, `switch`, `typedef`, `union`, `unsigned`, `void`, `volatile`, `while`.

Es recomendable seleccionar nombres de variables relacionadas al propósito de la variable. Es frecuente usar nombres cortos para variables locales, como los índices de un ciclo, y nombres más largos para variables externas.

Cada variable de un programa debe declararse antes de ser utilizada. La declaración consiste en enunciar el nombre de la variable y asociarle un tipo. La sintaxis correspondiente a la declaración de una variable es la siguiente:

```
[clase] tipo_C identificador;
```

donde

- `clase` determina si una variable tiene un carácter global (`static` o `extern`) o local (`auto` o `register`).
- `tipo_C` determina el tipo de datos de la variable (ver Tabla 2.1).
- `identificador` indica el nombre de la variable.

Toda variable declarada fuera de las funciones tiene ámbito *global*, es decir, puede ser accedida desde cualquier parte del programa. Por el contrario, una variable dentro de una función es por defecto *local*, es decir, sólo puede ser accedida dentro de la función donde se declaró.

En el Programa 2 se muestra un ejemplo de la declaración de variables y operaciones de suma con ellas.

Programa 2: Declaración de variables.

```
1  #include <stdio.h>
2
3  int main (void) {
4    typedef int ENTERO;
5    float valor_f1 = 1.2389, valor_f2 = 5.4321, sumaf;
6    ENTERO valor_i1 = 5, valor_i2 = 7, sumai;
7
8    sumai = valor_i1 + valor_i2;
9    sumaf = valor_f1 + valor_f2;
10
11   printf("%d + %d = %07d \n", valor_i1, valor_i2, sumai);
12   printf("%2.2f + %2.4f = %10.1f \n", valor_f1, valor_f2, sumaf);
13
14   return 0;
15 }
```

En las línea `9` se declara la función `main` de tipo entero y no admite argumentos de entrada. En las línea `11` se define un tipo de dato `ENTERO` como un sinónimo del tipo `int`. En las líneas `12` y `13` se declaran variables enteras y flotantes. En las líneas `15` y `16` se hacen las operaciones de suma de las variables declaradas, mientras que en las líneas `18` y `19` se imprimen los resultados de las sumas. Finalmente, la línea `21` retorna un cero, debido a que la función `main` se declaró de tipo entero. La salida de este programa en pantalla será:

```
5 + 7 = 0000012
1.24 + 5.4321 =        6.7
```

Ejemplo 2.1: (5 min)

Escribir y ejecutar el Programa 2 en tu sistema.

2.3. Constantes

Una *constante* es un valor que, una vez fijado por el compilador, no cambia durante la ejecución del programa. Una constante en C puede ser un número, un carácter o una cadena de caracteres. En un programa en C, se pueden definir constantes de dos formas:

1. Usando la directiva `#define`.
2. Usando la palabra clave `const`.

Los Programas 3 y 4 muestran dos programas que son funcionalmente idénticos, ambos despliegan en pantalla los siguientes valores de constantes definidas mediante `#define` y `const`:

```
Constante entera: 10
Constante de punto flotante: 4.5
Constante caracter: G
Cadena constante: ABC
```

Programa 3: Declaración de constantes usando `#define`.

```c
#include <stdio.h>

#define val  10
#define floatVal  4.5
#define charVal 'G' // Comillas simples
#define stringVal "ABC" // Comillas dobles

int main(void)
{
    printf("Constante entera: %d\n",val);
    printf("Constante de punto flotante: %.1f\n",floatVal);
    printf("Constante caracter: %c\n",charVal);
    printf("Cadena constante: %s\n",stringVal);

    return 0;
}
```

Programa 4: Declaración de constantes usando `const`.

```c
#include <stdio.h>

int main()
{
    const int val = 10;
    const float floatVal = 4.5;
    const char charVal = 'G';
    const char stringVal[10] = "ABC";

    printf("Constante entera: %d\n",val);
    printf("Constante de punto flotante: %.1f\n",floatVal);
    printf("Constante caracter: %c\n",charVal);
    printf("Cadena constante: %s\n",stringVal);

    return 0;
}
```

Ejemplo 2.2: (5 min)

Escribir y ejecutar los Programas 3 y 4 en tu sistema.

2.4. Funciones de entrada/salida

Antes de continuar con los operadores y expresiones en C, se hará una breve revisión de dos funciones importantes para la entrada y salida de datos, `scanf` y `printf`, que forman parte de la biblioteca estándar `stdio.h`. Estas funciones permitirán al programa comunicarse con el exterior, mediante el ingreso de datos por teclado e impresión en pantalla de resultados, lo cual será útil en los ejercicios subsecuentes.

La función `printf` permite escribir bytes en `stdout` (monitor o dispositivo de salida estándar) usando formatos específicos que se muestran en la Tabla 2.2:

```
printf("Hola mundo \n");
printf("Imprimir %c %d %f \n",'a',28,3.0e+8);
```

Tipo	Formato	Salida
Numérico	`%d`	(`int`) entero con signo base 10
	`%i`	(`int`) entero con signo base 10
	`%u`	(`int`) entero sin signo base 10
	`%o`	(`int`) entero sin signo base 8
	`%x`	(`int`) entero sin signo base 16 (`0,1,...,9,a,b,...,f`)
	`%X`	(`int`) entero sin signo base 16 (`0,1,...,9,A,B,...,F`)
	`%f`	(`double`) valor con signo de la forma `[−]dddd.dddd`
	`%e`	(`double`) valor con signo de la forma `[−]d.dddde[±]ddd`
	`%E`	(`double`) valor con signo de la forma `[−]d.ddddE[±]ddd`
	`%g`	(`double`) valor con signo en formato `f` ó `e`
	`%G`	(`double`) igual que `G`, pero introduce `E` en vez de `e`
Carácter	`%c`	carácter simple
	`%s`	cadena de caracteres
	`%%`	imprime el carácter de porcentaje (`%`)

Tabla 2.2: Especificadores de formato para la función `printf`.

Un *especificador de precisión* combina un punto y un entero para indicar el número de posiciones decimales para los números de punto flotante y se puede colocar justo después del *especificador del ancho mínimo del campo*. Por ejemplo, `%10.3f` asegura que la longitud del ancho mínimo del campo sea de 10 caracteres con 3 posiciones decimales.

Una *secuencia de escape* siempre representa a un carácter del ASCII. Son una combinación de barra invertida '\' y un carácter, y se clasifican en:

- **Gráficos** que corresponden con signos de puntuación usados para escribir.
- **No gráficos** que representan acciones, por ejemplo, mover el cursor de la pantalla al principio de la línea siguiente.

Estas secuencias parecen dos caracteres, aunque representan a uno solo. En la Tabla 2.3 se listan las secuencias de escape utilizadas por la función `printf`.

Secuencia	Descripción
`\n`	Nueva línea. El cursor pasa a la primera posición de la siguiente línea.
`\r`	Retorno de carro. El cursor pasa a la primera posición de la línea actual.
`\t`	Tabulador. El cursor pasa a la siguiente posición de tabulación.
`\a`	Alarma. Emite un sonido beep.
`\b`	Espacio atrás. Retrocede el cursor una posición a la izquierda.
`\f`	Alimentación de página. Crea una nueva página.
`\v`	Tabulación vertical.
`\\`	Muestra la barra invertida.
`\"`	Muestra la doble comilla.
`\'`	Muestra una comilla simple.
`\?`	Muestra el carácter de interrogante.

Tabla 2.3: Secuencias de escape utilizadas por la función `printf`.

La función `scanf` permite la entrada de bytes (caracteres ASCII) desde el `stdin` (teclado o dispositivo de entrada estándar), los interpreta de acuerdo a un formato especificado y los almacena en sus respectivas variables:

```
scanf("%f",&flotante);
scanf("%c\n",&caracter);
scanf("%f %d %c",&flotante, &entero, &caracter);
scanf("%ld",&entero_largo);
scanf("%s",cadena); //No lleva &
```

Si se usa el especificador `%s` para leer una cadena de caracteres, `scanf` leerá caracteres hasta encontrar un espacio, enter, tabulador, o retorno de carro. La cadena de caracteres se guarda en un arreglo que debe ser lo suficientemente grande como para almacenarla. Al final añade automáticamente el carácter nulo '`\0`'.

El Programa 5 muestra un código para usar las funciones `scanf` y `printf` para leer datos de diferente tipo desde el teclado y deplegarlos en pantalla.

Programa 5: Uso de las funciones `printf` y `scanf`.

```c
#include <stdio.h>
#define TAM_MAXIMO 80

int main(void)
{
  char cadena[TAM_MAXIMO];
  int x, y;
  float z;

  printf("Introduzca dos enteros separados por un espacio: ");
  scanf("%d %d",&x, &y);

  printf("Introduzca un valor flotante: ");
  scanf("%f",&z);

  printf("Introduzca una cadena: ");
  scanf("%s",cadena);

  printf("Usted introdujo: ");
  printf("%d %d\t%.3f\t%s\n",x,y,z,cadena);

  return 0;
}
```

Ejemplo 2.3: (5 min)

Escribir y ejecutar el Programa 5 en tu sistema.

Ejemplo 2.4: (5 min)

Escribir y ejecutar en tu sistema un programa que pida tu nombre, fecha de nacimiento y edad, y despliegue esta información en pantalla.

2.5. Operadores y expresiones

Los *operadores* son símbolos que indican la forma en cómo se manipulan los datos. Se aplican a variables u otros objetos denominados *operandos*. Los operadores

se pueden clasificar en los siguientes grupos: aritméticos, lógicos, relacionales, unarios, lógicos para manejo de bits, de asignación, incremento y decremento, operador condicional, entre otros.

2.5.1. Operadores aritméticos

Los operadores binarios[†] para cálculos aritméticos se listan en la Tabla 2.4.

Operador	Operación	Descripción	Ejemplo
`+`	Suma	Los operandos pueden ser enteros o flotantes	`5+2→7`
`-`	Resta		`5-2→3`
`*`	Multiplicación		`5*2→10`
`/`	División		`5/2→2.5`
`%`	Módulo[‡]	Los operandos deben ser enteros	`5%2→1`

[‡]Resto de la división: $a\,\%b = a - (\lfloor a/b \rfloor \times b)$

Tabla 2.4: Operadores aritméticos.

Programa 6: Operadores aritméticos.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int a=10, b=3, c;
6    float w=2.0, x=3.77, y, z;
7
8    c = w-x; // La resta es -1 de tipo entero
9    y = a+b; // La suma es 13.0 de tipo flotante
10   z = a*x; // La multiplicacion es 37.7 de tipo flotante
11
12   return 0;
13 }
```

En el Programa 6 se muestra un código con diferentes operaciones aritméticas entre datos de diferente tipo. El resultado siempre será del tipo de dato de la variable a donde se asignará. En la línea `8` se hace la resta de dos números flotantes, por lo que se espera un resultado flotante, es decir, $2.0 - 3.77 = -1.77$; sin embargo, el resultado se asigna a una variable entera, de modo que únicamente se preserva

[†]Un operador binario involucra a dos operandos.

la parte entera, es decir, -1. En la línea `9` se hace la suma de dos números enteros y el resultado se asigna a un flotante; por tanto, la suma es de tipo flotante. En la línea `10` se multiplica un número entero por un flotante y el resultado se asigna a un flotante; por tanto, la multiplicación es de tipo flotante (ver Sección 2.6).

Ejemplo 2.5: (5 min)

Escribir y ejecutar en tu sistema el Programa 6 y que despliegue en pantalla lo siguiente:

```
2.0 - 3.77 = -1
10 + 3 = 13.0
10 * 3.77 = 37.7
```

2.5.2. Operadores lógicos

Los operadores lógicos actúan sobre operandos lógicos que se interpretan como `FALSO` (**0**) y `VERDADERO` (**1**). Hay de tipo unario[†] y binario, como se muestra en la Tabla 2.5.

Tipo	Operador	Operación
Unario	`!`	Negación (NOT)
Binario	`&&`	Conjunción (AND)
	`||`	Disyunción (OR)

Tabla 2.5: Operadores lógicos.

El resultado de los operadores lógicos es de tipo `int`. Los operandos pueden ser enteros, reales o punteros. El resultado de la negación (NOT) es **0** si el operando tiene un valor distinto de cero, y **1** en caso contrario. El resultado de la conjunción (AND) da como resultado **1** si ambos operadores son distintos de cero; si uno de ellos es cero el resultado es **0**. La disyunción (OR) da como resultado **0** si ambos operandos son cero; si uno de los operandos tiene un valor distinto de cero, el resultado es **1**. Algunas de estas operaciones se ejemplifican en el Programa 7.

[†]Un operador unario involucra a un sólo operando.

Programa 7: Operadores lógicos.

```c
#include <stdio.h>

int main(void)
{
    int p = 10, q = 0, and, or, not;

    and = p&&q; // El resultado es 0
    or = p||q; // El resultado es 1
    not = !p; // El resultado es 0

    return 0;
}
```

Ejemplo 2.6: (5 min)

Escribir y ejecutar en tu sistema el Programa 7 y que despliegue en pantalla lo siguiente:

```
10 && 0 = 0
10 || 0 = 1
!10 = 0
```

2.5.3. Operadores relacionales

Los operadores relacionales se utilizan para formar expresiones lógicas `FALSO` (**0**) y `VERDADERO` (**1**). Los operandos pueden ser de tipo entero, real o puntero. En la Tabla 2.6 se listan los operadores relacionales en C.

Una expresión Booleana da como resultado los valores lógicos **0** y **1**. Los operadores que intervienen en una expresión Booleana pueden ser lógicos y relacionales, como se muestra en el Programa 8.

Operador	Operando	Ejemplo a=1 y b=2
>	a mayor que b	a>b→0
>=	a mayor o igual b	a>=b→0
<	a menor que b	a<b→1
<=	a menor o igual que b	(a+1)<=b→1
==	a igual que b	a==b→0
!=	a diferente que b	(a+b)!=3→0

Tabla 2.6: Operadores relacionales.

Programa 8: Expresiones Booleanas.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int p, q;
6    float x=15, y=18, z=20;
7
8    p = x==y; // p = 0
9    q = (x<y)&&(y<=z); // q = 1
10
11   return 0;
12 }
```

2.5.4. Operadores lógicos para manejo de bits

Los operadores lógicos para manejo de bits ejecutan operaciones bit-a-bit. En C existen seis operadores para la manipulación de bits, mostrados en la Tabla 2.7(a), los cuales sólo pueden ser aplicados a datos de tipo entero (`char`, `int`, `short`, `long`), no pueden ser flotantes. La tabla de verdad para los operadores AND (`&`), OR (`|`), y XOR (`^`) se muestra en la Tabla 2.7(b).

Suponga dos números `a=5` y `b=9`; en formato binario (1 byte) estos valores son `a=00000101` y `b=00001001`. Las operaciones entre los bits para las variables de este ejemplo son (ver Programa 9):

- `a&b=00000001`: la operación AND se usa frecuentemente para *enmascarar* o *apagar* (pasar de **1** a **0**) un conjunto de bits.

Operador	Descripción
`&`	Operación AND a nivel de bits
`|`	Operación OR a nivel de bits
`^`	Operación XOR a nivel de bits
`<<`	Desplazamiento a la izquierda
`>>`	Desplazamiento a la derecha
∼	Complemento a uno (unario)

(a)

p	q	p&q	p|q	p^q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

(b)

Tabla 2.7: (a) Operadores lógicos para manipulación de bits. (b) Tabla de verdad.

- `a|b=00001101`: la operación OR se usa para encender (pasar de **0** a **1**) un conjunto de bits.
- `a^b=00001100`: la operación OR exclusiva (XOR) se usa para encender cada bit en donde sus respectivos operandos son diferentes, y apagar cada bit en donde sus correspondientes operandos son iguales.
- `b<<1=00010010` y `b>>1=00000100`: realiza desplazamientos a la izquierda y derecha del operando a la izquierda por un número de posiciones dado por el operando a la derecha, en este caso, una posición. En ambos casos se rellenan las posiciones vacantes con ceros.
- ∼`a = 11111010`: realiza el complemento a uno, es decir, convierte cada bit en **0** a **1** y viceversa.

Decimal	Hex	Binario
0	0	0000
1	1	0001
2	2	0010
3	3	0100
4	4	1000
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Programa 9: Demostración del uso de operados lógicos para manejo de bits. Los resultados se presentan en formato hexadecimal para facilitar su interpretación.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      unsigned char a = 5; // a = 00000101
6      unsigned char b = 9; // b = 00001001
7
8      printf("a = %X, b = %X\n", a, b);
9      printf("a&b = %X\n", a&b);  // 00000001 (0x1)
10     printf("a|b = %X\n", a|b);  // 00001101 (0xD)
11     printf("a^b = %X\n", a^b);  // 00001100 (0xC)
12     printf("~a = %X\n", a = ~a); // 11111010 (0xFA)
13     printf("b<<1 = %X\n", b<<1); // 00010010 (0x12)
14     printf("b>>1 = %X\n", b>>1); // 00000100 (0x4)
15
16     return 0;
17 }
```

2.5.5. Operadores de asignación

El *operador de asignación* más básico se representa con el símbolo igual '=' y la sintaxis de asignación es

```
identificador = expresión;
```

Además, expresiones como

```
i = i + 2
```

en donde la variable a la izquierda es repetida inmediatamente en la derecha, puede escribirse de la siguiente forma compacta

```
i += 2
```

donde el operador '+=' también es de asignación. En la Tabla 2.8 se listan los operadores de asignación en C.

Operador	Descripción	Equivalente
x=y	Asignación	x=y
x+=y	Suma y asignación	x=x+y
x-=y	Resta y asignación	x=x-y
x*=y	Multiplicación y asignación	x=x*y
x/=y	División y asignación	x=x/y
x%=y	Módulo y asignación	x=x%y
x<<=y	Desplazamiento a la izquierda y asignación	x=x<<y
x>>=y	Desplazamiento a la derecha y asignación	x=x>>y
x&=y	AND bit-a-bit y asignación	x=x&y
x|=y	OR bit-a-bit y asignación	x=x|y
x^=y	XOR bit-a-bit y asignación	x=x^y

Tabla 2.8: Operadores de asignación.

Sean `expr1` y `expr2` las expresiones 1 y 2, y `op` un operador (como `+`), entonces

```
expr1 = op= expr2
```

es equivalente a

```
expr1 = (expr1) op (expr2)
```

donde `expr1` se computa una sola vez. Nótese los paréntesis que encierran `expr2`. Por ejemplo,

```
x *= x + 1
```

significa

```
x = x  *  (x + 1)
```

en lugar de

```
x = x  *  x + 1
```

2.5.6. Operadores de incremento y decremento

El lenguaje C provee dos operadores unarios para incrementar y decrementar variables. El operador de incremento '`++`' suma 1 a su operando, mientras que el operador de decremento '`--`' sustrae 1.

Ambos operadores pueden ser prefijos (antes de la variable, `++x`) o sufijos (después de la variable, `x++`). En ambos casos el efecto es incrementar o decrementar la variable, aunque un operador prefijo (`++x`) modifica la variable antes de ser utilizada, mientras que un operador sufijo (`x++`) modifica la variable después de ser utilizada. Por ejemplo, si `n=1`, entonces

```
x = n++;
```

se tendrá `x=1` y `n=2`, y en el caso

```
x = ++n;
```

se tendrá `x=2` y `n=2`. Nótese que en ambos casos `n` se convierte en `2`. Los operadores de incremento o decremento sólo aplican a variables; una expresión como `(i+j)++` es ilegal.

2.5.7. Expresión condicional

La *expresión condicional*, escrita con el operador ternario[†] '`?:`', provee una alternativa para escribir construcciones condicionales de forma compacta. En la expresión

[†]Un operador ternario involucra a tres operandos.

```
expr1 ? expr2 : expr3
```

la expresión `expr1` se evalúa primero. Si es `VERDADERA`, entonces la expresión `expr2` es evaluada y ese es el valor de la expresión condicional, en otro caso, la expresión `expr3` es evaluada y ese es el valor de salida. Por ejemplo, la construcción condicional

```
if (a > b)
        x = a;
else
        x = b;
```

computa en `x` el máximo de `a` y `b`, puede ser escrita como la expresión condicional

```
x = (a > b) ? a : b; /* x = max(a, b) */
```

2.5.8. Otros operadores

- **Operador de coma** (`,`): sirve para separar expresiones, las cuales se evalúan de izquierda a derecha. Por ejemplo: `aux = v1, v1 = v2, v2 = aux;`
- **Operador de indirección** (`*`): accede a un valor indirectamente a través de un puntero. El resultado es el valor direccionado por el operando.
- **Operador de dirección–de** (`&`): provee la dirección de su operando. Por ejemplo:

  ```
  int *pa, b; // pa es un puntero a un valor entero
  int a[10]; // a es un arreglo de 10 elementos de tipo entero
  pa = &a[6]; // en el puntero pa se almacena la dirección
              // del séptimo elemento del arreglo a
  b = *pa; // se le asigna a b el valor almacenado en la
           // dirección especificada por pa
  ```

- **Operador `sizeof`** (tamaño de): provee el tamaño en bytes de su operando, cuyo resultado es un entero sin signo. Este operador se puede aplicar a cualquier tipo de datos, incluidos los tipos básicos, como los tipos enteros y de punto flotante, los tipos de punteros o los tipos de datos compuestos como estructura, unión, etc. Por ejemplo:

```
printf("%lu\n",sizeof(char));
printf("%lu\n",sizeof(int));
printf("%lu\n",sizeof(float));
printf("%lu\n", sizeof(double));
```

imprime `1`, `4`, `4`, y `8`, que corresponde al tamaño en bytes de un `char`, `int`, `float`, y `double`, respectivamente (ver Tabla 2.1).

2.5.9. Precedencia y asociatividad

La *precedencia* especifica la prioridad de las operaciones en las expresiones que contienen más de un operador. La *asociatividad* especifica si, en una expresión que contiene varios operadores con la misma prioridad, un operando se agrupa con el de su izquierda o el de su derecha.

En la Tabla 2.9 se resumen las reglas de precedencia y asociatividad de todos los operadores en C, incluidos algunos que no se han revisado. Los operadores escritos sobre un mismo renglón tienen la misma prioridad. Los renglones se han ordenado de mayor a menor prioridad. Una expresión entre paréntesis siempre se evalúa primero. Los paréntesis tiene la mayor prioridad y son evaluados de más internos a más externos.

Nivel	Precedencia	Asociatividad
1	`() [] ->.`	izquierda a derecha
2	`! ~ ++ -- * & (tipo) sizeof`	derecha a izquierda
3	`* / %`	izquierda a derecha
4	`+ -`	izquierda a derecha
5	`<< >>`	izquierda a derecha
6	`< <= > >=`	izquierda a derecha
7	`== !=`	izquierda a derecha
8	`&`	izquierda a derecha
9	`^`	izquierda a derecha
10	`|`	izquierda a derecha
11	`&&`	izquierda a derecha
12	`||`	izquierda a derecha
13	`?:`	derecha a izquierda
14	`= += -= *= /=%= &= ^= |= <<= >>=`	derecha a izquierda
15	`,`	izquierda a derecha

Tabla 2.9: Precedencia y asociatividad de los operadores.

La operaciones matemáticas tienen un orden de precedencia, de modo que algunas operaciones se resuelven antes que otras, por lo que un paréntesis en un lugar diferente de las operaciones puede dar lugar a distintos resultados. Por ejemplo:

- **Desarrollo 1**:

```
a = 12/3+2*2-1
  = 4+4-1
  = 8-1
  = 7
```

- **Desarrollo 2**:

```
a = 12/(3+2)*2-1
  = 12/5*2-1
  = 2.4*2-1
  = 4.8-1
  = 3.8
```

- **Desarrollo 3**:

```
a = (12/3)+2*(2-1)
  = 4+2*1
  = 4+2
  = 6
```

- **Desarrollo 4**:

```
a = 12/(3+2*(2-1))
  = 12/(3+2*1)
  = 12/(3+2)
  = 12/5
  = 2.4
```

Ejemplo 2.7: (20 min)

1) Determinar el resultado de las siguientes operaciones:

```
int a = 5 * 3 + 2 - 4; // a = 13
int a = 2 + 4 + 3 * 5 / 3 - 5; // a = 6
double b = 5 % 3 & 4 + 5 * 6; // b = 2.0
double b = 5 & 3 && 4 || 5 | 6; // b = 1.0
```

2) Verificar tus resultados escribiendo un programa.

2.6. Conversión de tipos

2.6.1. Implícita

Cuando los operando dentro de una expresión son de tipos diferentes, se convierten automáticamente a un tipo común, de acuerdo con ciertas reglas. En general, las únicas conversiones automáticas son aquellas que convierten un operando 'angosto' (menor precisión) a uno más 'ancho' (mayor precisión) sin pérdida de información, como se muestra en la Figura 2.1.

Entonces las reglas de *conversión implícita* son las siguientes:

1. Cualquier operando de tipo `float` es convertido a tipo `double`.
2. Si un operando es de tipo `long double`, el otro operando es convertido a `long double`.
3. Si un operando es de tipo `double`, el otro operando es convertido a `double`.
4. Cualquier operando de tipo `char` o `short` es convertido a tipo `int`.

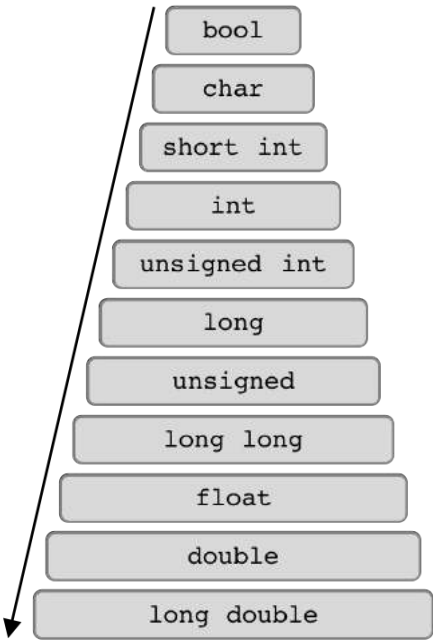


Figura 2.1: Conversión implícita del tipo de dato.

5. Cualquier operando de tipo `unsigned char` o `unsigned short` es convertido a tipo `unsigned int`.
6. Si un operando es de tipo `unsigned long`, el otro operando es convertido a `unsigned long`.
7. Si un operando es de tipo `long`, el otro operando es convertido a `long`.
8. Si un operando es de tipo `unsigned int`, el otro operando es convertido a `unsigned int`.

Considere el siguiente ejemplo:

```
long a;
unsigned char b;
int c;
float d;
int f;
f = a + b * c / d;
```

Teniendo en cuenta que se realiza primero la multiplicación, después la división y por último la suma, la conversión de tipos se realizaría como:

1. `b` es convertido a `unsigned int` (regla 5).
2. `c` es convertido a `unsigned int` (regla 8). Se realiza la multiplicación (`*`) y se obtiene el resultado de tipo `int`.
3. `d` es convertido a `double` (regla 1).
4. El resultado de `b*c` es convertido a `double` (regla 3). Se realiza la división (`/`) y se obtiene el resultado de tipo `double`.
5. `a` es convertido a `double` (regla 3). Se realiza la suma (`+`) y se obtiene un resultado de tipo `double`.
6. El resultado de `a + b * c / d`, para ser asignado a `f`, es convertido a entero por truncamiento, esto es, eliminando la parte fraccionaria.

En resumen:

- Los operandos que intervienen en una determinada operación, son convertidos al tipo de precisión más alta.
- En C, la aritmética de punto flotante se realiza en doble precisión.
- En una asignación, el valor de la parte derecha es convertido al tipo del valor de la parte izquierda.

2.6.2. Explícita

En C, está permitido una *conversión explícita* del tipo de una expresión mediante una construcción denominada *cast*, que tiene la forma:

```
(nombre-de-tipo)expresión;
```

La expresión es convertida al tipo especificado aplicando las reglas de conversión expuestas anteriormente. Por ejemplo,

```
int x = 5;
float y;
y = x/2;
```

El valor asignado a la variable `y` será `2.0`, ya que se realiza una división entera debido a que `x` es de tipo entero y se trunca para asignarlo a `y` que es de tipo

flotante. Si se desea obtener el valor exacto de la división, se debe realizar la conversión de `x` a flotante:

```
int x = 5;
float y;
y = (float)x/2;
```

en este caso el valor asignado a la variable `y` será `2.5`.

3. Estructuras y sentencias de control

Las *estructuras de control* permiten modificar el flujo de ejecución de las instrucciones de un programa. En C existen tres tipos de estructuras de control:

1. **Secuenciales**: ejecución sucesiva de dos o más operaciones.
2. **Selectivas**: se realiza una u otra operación dependiendo de una condición.
3. **Iterativas**: repetición de una operación mientras no se cumpla una condición.

3.1. Estructuras selectivas

Las *estructuras de selección* (o bifurcación) se utilizan para elegir entre diversos cursos de acción. En lenguaje C existen tres tipos de estructuras de selección: `if-else`, `else if`, y `switch`.

3.1.1. Sentencia `if-else`

Se toma una decisión referente a la acción a ejecutar en un programa, basándose en el resultado lógico (`VERDADERO` o `FALSO`) de una expresión. La sintaxis de la estructura `if-else` es

```
if (expresión)
        sentencia1;
else
        sentencia2;
```

donde la cláusula `else` es opcional. De esta estructura selectiva se derivan tres casos:

1. Si el resultado de la `expresión` es `VERDADERO`, se ejecutará lo indicado por la `sentencia1`.
2. Si el resultado de la `expresión` es `FALSO`, se ejecutará lo indicado por la `sentencia2`.
3. Si el resultado de la `expresión` es `FALSO` y `else` se ha omitido, se ignorará la `sentencia1`.

En cualquier caso, el programa continua con la siguiente sentencia ejecutable.

Debido a que `if` simplemente prueba el valor numérico de una expresión, la forma más obvia de escribirlo es

```
if (expresión)
```

en lugar de

```
if (expresión != 0)
```

Por ejemplo, en el código

```
if (x)
   b = a / x;
b = b + 1;
```

la *expresión* es un valor numérico `x`. Entonces `b=a/x`, que representa la `sentencia1`, se ejecutará si la expresión es verdadera (`x` es distinta de cero), y no se ejecutará si la expresión es falsa (`x` es igual de cero). En cualquier caso se continua la ejecución en la línea siguiente, `b=b+1`.

Es posible anidar las sentencias `if-else`, esto quiere decir que se puede escribir una secuencia `if-else` dentro de otra secuencia `if-else`:

```
if (expresión1)
  if (expresión2)
    sentencia1;
  else
    sentencia2;
```

expr1	expr2	sent1	sent2
F	F	no	no
F	V	no	no
V	F	no	si
V	V	si	no

En este ejemplo nótese que existe un sólo `else` en una secuencia anidada que llama a dos `if`. Para evitar ambigüedades cuando un `else` es omitido de una secuencia anidada `if`, se asocia el `else` con el `if` previo más cercano. En el ejemplo anterior, el `else` forma parte de la secuencia `if-else` asociada a la `expresión2`.

Otra manera de evitar ambigüedades es usar llaves '`{ }`' para delimitar explícitamente el bloque de cada secuencia `if-else`, por ejemplo:

```
if (expresión1)
{
  if (expresión2)
    sentencia1;
}
else
  sentencia2;
```

expr1	expr2	sent1	sent2
F	F	no	si
F	V	no	si
V	F	no	no
V	V	si	no

lo cual indica que el `else` forma parte de la secuencia `if-else` asociada a la `expresión1`. En el Programa 10 se muestra el uso de secuencias `if-else` anidadas para determinar el menor de tres números.

Programa 10: Menor de tres números.

```c
#include <stdio.h>

int main(void) {
  float a, b, c, menor;

  printf("Introduce tres numeros a, b y c: ");
  scanf("%f %f %f", &a, &b, &c);

  if (a < b)
    if (a < c)
      menor = a;
    else
      menor = c;
  else
    if (b < c)
      menor = b;
    else
      menor = c;

  printf("El menor es %f\n", menor);
  return 0;
}
```

Ejemplo 3.1: (5 min)

Modifica el Programa 10 para que determine el mayor de tres números.

3.1.2. Sentencia `else if`

La construcción

```
if (expresión1)
  sentencia1;
else if (expresión2)
  sentencia2;
else if (expresión3)
  sentencia3;
.
.
.
else
  sentenciaN;
```

es bastante utilizada para escribir programas con múltiples opciones de decisión. Las `expresiones` son evaluadas en orden; si cualquiera de ellas es verdadera, la `sentencia` asociada se ejecuta y esto termina toda la cadena de opciones. Como siempre, el código para cada `sentencia` puede ser una sola línea de código o varias sentencias agrupadas entre llaves. El último `else` indica 'ninguna de las anteriores', es decir, el caso por default donde ninguna de las opciones anteriores se satisfizo. En algunos casos el `else` puede ser omitido, o puede ser usado como un verificador de error para detectar una condición 'imposible'.

En el Programa 11 se muestra un ejemplo del uso de la estructura `else-if` para calcular el importe que se debe pagar de acuerdo a varias opciones de descuento en función de la cantidad comprada.

Programa 11: Cantidad a pagar en función de la cantidad comprada.

```c
#include <stdio.h>

int main(void)
{
  char ar[20];
  int cc;
  float pu;

  printf("Nombre del articulo: ");
  scanf("%s", ar);
  printf("Cantidad comprada: ");
  scanf("%d", &cc);
  printf("Precio unitario: ");
  scanf("%f", &pu);

  printf("\n\n%20s\t%10s\t%10s\t%10s\t%10s\n",
         "Articulo","Cantidad","P.U.","%","Total");
  printf("%20s\t%10d\t%10.2f", ar, cc, pu);

  if (cc > 100)
    printf("\t%9d%%\t%10.2f\n\n", 40, cc*pu*0.6);
  else if (cc >= 25)
    printf("\t%9d%%\t%10.2f\n\n", 20, cc*pu*0.8);
  else if (cc >= 10)
    printf("\t%9d%%\t%10.2f\n\n", 10, cc*pu*0.9);
  else
    printf("\t%10s\t%10.2f\n\n", "--", cc*pu);
  return 0;
}
```

Ejemplo 3.2: (5 min)

Escribir y ejecutar en tu sistema el Programa 11.

3.1.3. Sentencia `switch`

La sentencia `switch` está orientada a múltiples opciones para probar si una expresión coincide con alguno de los valores constantes enteros y se bifurca de acuerdo a esto:

```
switch (expresion) {
  case const-expr1:
    setencia1;
  case const-expr2:
    setencia2;
  .
  .
  .
  default:
    setenciaN;
}
```

Cada `case` está etiquetado con un valor entero constante o una expresión constante. Si un caso coincide, se comienza la ejecución de ese caso. Todas la expresiones de `case` deben ser diferentes. El caso etiquetado como `default` se ejecutará si ninguno de los anteriores se satisfizo. Un `default` es opcional; si se omitió y ninguno de los casos coincide, ninguna acción se ejecuta.

Debido a que los casos sólo sirven como etiquetas, después que el código de un `case` se realizó, la ejecución del programa continúa hacia el siguiente `case` a menos que se le indique una acción explícita de escape, lo cual se indica mediante las sentencias `break` o `return`. La sentencia `break` también puede ser utilizada para forzar la salida inmediata de ciclos `for`, `while` y `do`, como se discutirá en la Sección 3.2.4.

En el Programa 12 se muestra un ejemplo del uso de la sentencia `switch` para determinar si un carácter introducido por teclado es una vocal o una consonante.

Programa 12: Determina si una letra es vocal o consonante.

```c
#include <stdio.h>
int main(void) {
  char letra;
  printf("Introduzca una letra: ");
  scanf("%c",&letra);
  switch(letra) {
    case 'a':
    case 'A':
      printf("La letra %c es vocal\n",letra);
      break;
    case 'e':
    case 'E':
      printf("La letra %c es vocal\n",letra);
      break;
    case 'i':
    case 'I':
      printf("La letra %c es vocal\n",letra);
      break;
    case 'o':
    case 'O':
      printf("La letra %c es vocal\n",letra);
      break;
    case 'u':
    case 'U':
      printf("La letra %c es vocal\n",letra);
      break;
    default:
      printf("La letra %c es consonante\n",letra);
      break;
  }
  return 0;
}
```

Ejemplo 3.3: (5 min)

Escribir y ejecutar en tu sistema el Programa 12. Prueba qué sucede si se omite la sentencia `break`.

3.2. Estructuras iterativas

Las *estructuras iterativas* permiten repetir una serie de veces la ejecución de unas líneas de código durante un número determinado de veces o hasta que se cumpla una determinada condición de término. Estas estructuras se pueden anidar, aunque hay que estructurarlas de forma correcta. En lenguaje C existen tres tipos de estructuras iterativas: `for`, `while` y `do`.

3.2.1. Sentencia `for`

La sentencia `for` permite definir ciclos controlados por contador y se define de la siguiente manera:

```
for (inicialización; condición; progresión)
        sentencia;
```

`inicialización` define el estado inicial de un contador que controla la repetición del ciclo. `condición` es una expresión Booleana que define un criterio para realizar las repeticiones; si es `VERDADERA`, ejecuta la `sentencia`, se evalúa la expresión en `progresión`, y vuelve a evaluarse la `condición`; si es `FALSA`, finalizan las repeticiones y continua con la siguiente sentencia del programa. `progresión` es una expresión se utiliza para modificar el valor del contador en forma incremental o decremental. El código en `sentencia` puede ser una sola línea de código o varias sentencias agrupadas entre llaves.

En el siguiente ejemplo se imprimen los números del 1 al 100. Literalmente dice: desde `n` igual a `1`, mientras `n` sea menor o igual que `100`, con incrementos de `1`, imprime el valor de `n`.

```
int n; // contador
for (n = 1; n <= 100; n++)
   printf("%3d\n", n);
```

3.2.2. Sentencias `while` y `do`

La sentencia `while` se ejecutará mientras el valor de una condición sea verdadera y se define de la siguiente manera:

```
while (condición)
   sentencia;
```

`condición` puede ser una expresión relacional o lógica. Si la `condición` es `VERDADERA`, se ejecuta la sentencia y se vuelve a evaluar la `condición`; si es `FALSA`, la sentencia no se ejecuta y el programa continua. Nótese que la `condición` se evalúa primero, de modo que la `sentencia` puede nunca ejecutarse si la `condición` es `FALSA`. Se debe incluir algún elemento que altere el valor de `condición`, proporcionando la salida del ciclo. En el siguiente ejemplo se imprimen los números del 1 al 100. Literalmente dice: mientras `n` sea menor o igual que `100`, imprime el valor de `n` e incrementa en `1` el valor de `n`.

```
int n = 1; // contador
while (n <= 100) {
   printf("%3d\n", n);
   n++;
}
```

La sentencia `do` es similar al `while` excepto que la `condición` se evalúa al final del ciclo y no al inicio, lo que obliga a que se ejecute la `sentencia` por lo menos una vez. Se define de la siguiente manera:

```
do
   sentencia;
while (condición);
```

Al igual que en la sentencia `while`, se debe incluir algún elemento que altere el valor de `condición` para proporcionar eventualmente la salida del ciclo.

En el siguiente ejemplo se imprimen los números del 1 al 100. Literalmente dice: imprime el valor de `n` e incrementa su valor en `1` mientras `n` sea menor o igual que `100`.

```
int n = 1; // contador
do {
   printf("%3d\n", n);
   n++;
}
while(n<=100);
```

3.2.3. Ciclos anidados

En general, las sentencias `for`, `while` y `do` se pueden colocar indistintamente unas dentro de otras para formar ciclos anidados. Un ciclo interno se ejecutará totalmente cada vez que se ejecute el ciclo que lo contiene. A continuación se muestran ejemplos de cada estructura iterativa considerando dos ciclos anidados.

```
while(condición) {

  while(condición) {
    sentencia(s);
  }
  sentencia(s);
}
```

```
do {
  sentencia(s);

  do {
    sentencia(s);
  }while(condición);

}while(condición);
```

```
for (init;cond;incr) {

  for (init;cond;incr) {
    sentencia(s);
  }
  sentencia(s);
}
```

En el Programa 13 se muestra el ejemplo de ciclos `for` anidados para imprimir una tabla de multiplicar del 1 al 5.

Programa 13: Tabla de multiplicar del 1 al 5.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i, j;
6
7      for(i=1; i<=10; i++) {
8          for(j=1; j<=5; j++) {
9              printf("%d\t", (i*j));
10         }
11         printf("\n");
12     }
13     return 0;
14 }
```

Ejemplo 3.4: (10 min)

Modificar el Programa 13 para que trabaje con ciclos `while` anidados y otro código para que trabaje con ciclos `do` anidados.

3.2.4. Sentencias `break` y `continue`

La sentencia `break` provee una salida prematura de las estructuras iterativas `for`, `while` y `do`, de manera similar que en la sentencia `switch`. Además, si se están usando ciclos anidados, un `break` causa que el ciclo más interno termine inmediatamente y se ejecuta la siguiente línea después del bloque. Se usa casi siempre con una sentencia `if-else` dentro del ciclo. En la Figura 3.1 se muestra el funcionamiento de la sentencia `break` para las estructuras iterativas.

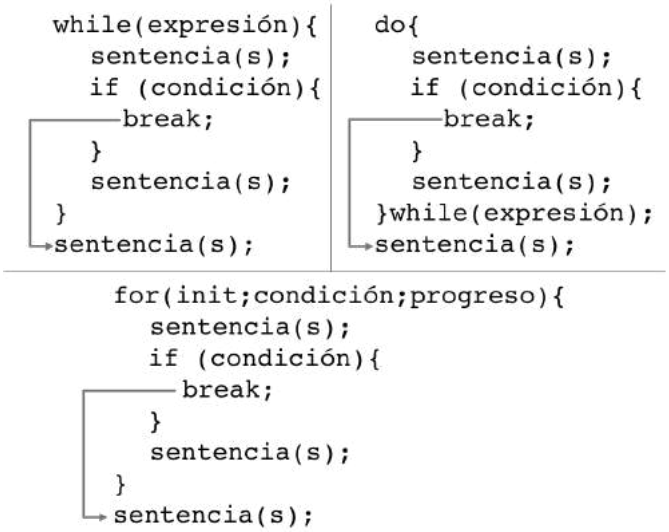


Figura 3.1: Funcionamiento de la sentencia `break` para las estructuras iterativas `while`, `do` y `for`.

En el Programa 14 se muestra un ejemplo del uso de la sentencia `break` dentro de un ciclo `do`. Se calcula la suma de un máximo de 10 números introducidos por teclado. Si un número negativo se introduce antes de completar los 10 números termina el ciclo y se muestra la suma.

Programa 14: Ejemplo del uso de la sentencia `break`.

```
1  #include <stdio.h>
2
3  int main(void) {
4    int i=1;
5    float num, suma = 0.0;
6    do {
7      printf("Ingresa n%d: ",i);
8      scanf("%f",&num);
9      i++;
10     if (num < 0)
11       break;
12     suma += num;
13   }while(i<=10);
14   printf("Suma = %.3f\n",suma);
15   return 0;
16 }
```

La sentencia `continue`, estando dentro de las estructuras iterativas `while`, `do` y `for`, pasa el control para que se ejecute la siguiente iteración, ignorando todas las sentencias dentro del ciclo que se encuentran después de la llamada al `continue`. Se usa casi siempre con una sentencia `if-else` dentro del ciclo. En la Figura 3.2 se muestra el funcionamiento de la sentencia `continue` para las estructuras iterativas.

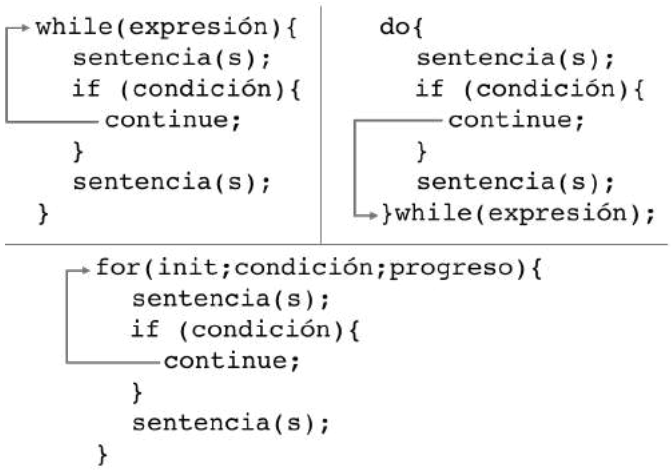


Figura 3.2: Funcionamiento de la sentencia `continue` para las estructuras iterativas `while`, `do` y `for`.

En el Programa 15 se muestra un ejemplo del uso de la sentencia `continue` dentro de un ciclo `for`. Se calcula la suma de un máximo de 10 números introducidos por teclado. Si un número negativo se introduce no se incluye en el cálculo de la suma.

Programa 15: Ejemplo del uso de la sentencia `continue`.

```
1  #include <stdio.h>
2
3  int main() {
4      int i;
5      float num, suma = 0.0;
6
7      for(i=1; i <= 10; ++i){
8          printf("Ingresa n%d: ",i);
9          scanf("%f", &num);
10         if (num < 0)
11             continue;
12         suma += num;
13     }
14     printf("Suma = %.3f\n", suma);
15     return 0;
16 }
```

Ejemplo 3.5: (10 min)

Escribir y ejecutar en tu sistema los Programas 14 y 15.

4. Estructuras de datos estáticas

Una *estructura de datos* es una colección de datos caracterizados por su organización y las operaciones que se definen sobre ella, como se muestra en la Figura 4.1. Se dividen en:

1. **Estructuras de datos estáticas:** el tamaño en memoria está predefinido y no puede modificarse durante la ejecución del programa.
2. **Estructuras de datos dinámicas:** no tienen limitaciones predefinidas en el tamaño de memoria ocupada.

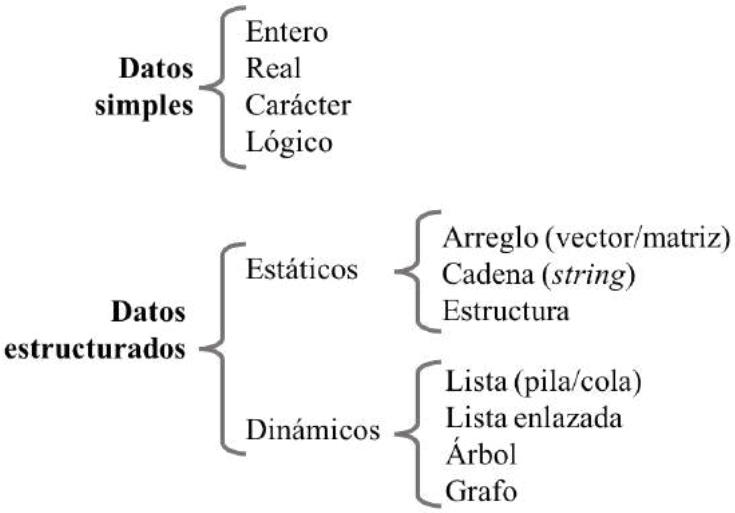


Figura 4.1: Estructuras de datos.

4.1. Arreglos

Las estructuras de datos básicas son los arreglos (*arrays*), que representan el concepto matemático de vector y matriz. Un arreglo es una secuencia de posiciones que tiene las siguientes características:

- Ordenado: el enésimo elemento puede ser identificado.
- Homogéneo: contiene datos del mismo tipo.
- Selección de posiciones mediante índices.
- Unidimensionales y multidimensionales.

4.1.1. Arreglos unidimensionales

Un *arreglo unidimensional* o vector consta de n elementos, donde cada uno tiene una posición específica en el vector, lo cual está designado por un *índice*. Por ejemplo, un arreglo unidimensional llamado `datos` con `n` elementos se organiza de la siguiente manera:

```
datos[0], datos[1], . . . , datos[i], . . . , datos[n-1]
```

Nótese que los índices son enteros consecutivos y el primer índice es `0`. En cada posición del vector se almacena un valor, por ejemplo, un vector llamado `X` con ocho elementos puede tener los siguientes valores:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]
14.0	12.0	8.0	7.0	6.41	5.23	6.15	7.25

La declaración de un arreglo unidimensional especifica el nombre del arreglo, número de elementos y tipo de datos:

```
tipo_C identificador[tamaño];
```

donde

- `tipo_C` indica el tipo de los elementos del arreglo (ver Sección 2.1). Puede ser cualquiera excepto `void`.
- `identificador` es el nombre del arreglo.
- `tamaño` es una constante que indica el número de elementos en el arreglo.

Por ejemplo, si se declaran

```
int lista[100];
char nombre[40];
```

se están creando dos arreglos, el primero llamado `lista` con 100 elementos (del 0 al 99), cada uno de ellos de tipo `int`, y el segundo llamado `nombre` con 40 elementos (del 0 al 39), cada uno de ellos de tipo `char`. En el Programa 16 se muestra la creación de un vector '`a`' con 10 elementos cuyos elementos son rellenados con valores introducidos por teclado.

Programa 16: Creación y rellenado de un vector.

```c
#include <stdio.h>
#define TAM 10

int main(void)
{
  int a[TAM], n = 0;

  do{
    printf("a[%d] = ", n);
    scanf("%d", &a[n]);
    n++;
  }while (n < TAM);
  return 0;
}
```

Ejemplo 4.1: (5 min)

Escribir y ejecutar en tu sistema el Programa 16. Además, agregar el código necesario para que imprima en pantalla el vector resultante.

4.1.2. Arreglos multidimensionales

Se pueden definir tablas o matrices como arreglos de varias dimensiones, los cuales pueden ser:

- **Arreglo bidimensional:** dos dimensiones, también llamadas matrices.
- **Arreglo multidimensional:** tres o más dimensiones, también llamadas hipermatrices.

De manera general, un *arreglo multidimensional* se declara como

```
tipo_C identificador[tam1][tam2]...[tamN];
```

donde

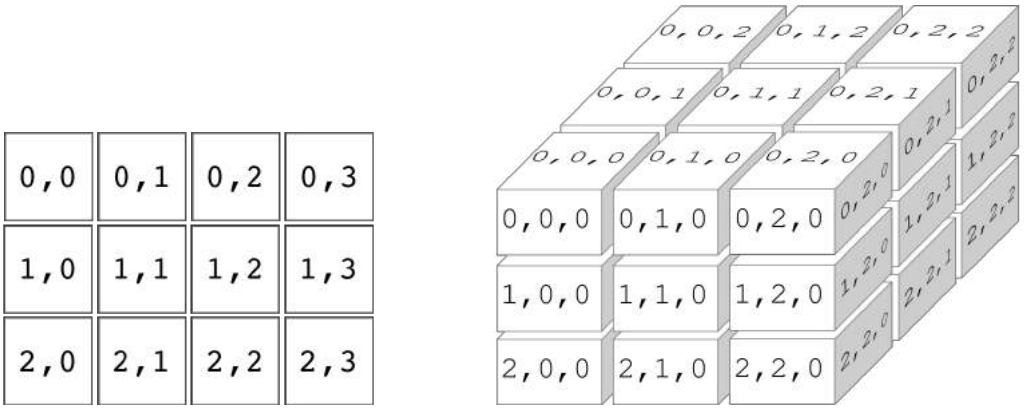
- `tipo_C` indica el tipo de los elementos del arreglo. Puede ser cualquiera excepto `void`.

- `identificador` es el nombre del arreglo.
- `tam1`, `tam2`, . . . , `tamN` son constantes que indican el número de elementos en cada dimensión del arreglo.

Por ejemplo, si se declaran

```
char array2d[3][4];
int array3d[3][3][3];
```

se declaran dos arreglos multidimensionales, el primero llamado `array2d` de dos dimensiones con 12 elementos (tres filas y cuatro columnas) todos de tipo `char` (ver Figura 4.2(a)), el segundo llamado `array3d` de tres dimensiones con 27 elementos todos de tipo `int` (ver Figura 4.2(b)).



(a) (b)

Figura 4.2: Arreglos multidimensionales: (a) en dos dimensiones de tamaño 3×4 y (b) en tres dimensiones de tamaño $3 \times 3 \times 3$.

Para acceder a un elemento del arreglo, se hace mediante el nombre del arreglo seguido de uno o más subíndices dependiendo de las dimensiones del mismo, cada uno de ellos encerrados entre corchetes '`[ ]`'; por ejemplo, `a[1]` accede a la segunda posición del vector `a`, o `b[1][2]` accede al elemento ubicado en la segunda fila y tercera columna de la matriz `b`. Un subíndice puede ser una constante o una variable.

Se pueden crear arreglos con valores preestablecidos desde su declaración. Por ejemplo:

```
int a[5] = {1,2,3,4,5};
int b[2][3] = {1,2,3,4,5,6};
```

se crean dos arreglos, el primero llamado 'a' es un vector con cinco elementos, mientras que el segundo es una matriz llamada 'b' con seis elementos. Debido a que los arreglos son almacenados por filas, el arreglo 'b' está organizado de la siguiente manera:

$$\texttt{b} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}.$$

Hay que señalar que el lenguaje C no verifica los límites del arreglo, es responsabilidad del programador realizar este tipo de operaciones.

En el Programa 17 se muestra la creación de una matriz `t` de tamaño 3×3 cuyos elementos son rellenados con valores introducidos por teclado.

Programa 17: Creación y rellenado de una matriz.

```
1  #include <stdio.h>
2  #define FILAS 3
3  #define COLS 3
4
5  int main(void)
6  {
7    int t[FILAS][COLS];
8    int f, c;
9
10   for(f=0; f<FILAS; f++)
11   {
12     printf("\nDatos para la fila %d\n",f);
13     for(c=0; c<COLS; c++)
14       scanf("%d", &t[f][c]);
15   }
16   return 0;
17 }
```

Ejemplo 4.2: (5 min)

Escribir y ejecutar en tu sistema el Programa 17. Además, agregar el código necesario para que imprima en pantalla la matriz resultante.

4.2. Cadenas de caracteres

Una *cadena de caracteres* es un arreglo unidimensional en el cual todos sus elementos son de tipo `char`. La declaración de una cadena de caracteres debe especificar la longitud del arreglo, por ejemplo:

```
char nombre[40];
```

declara un arreglo llamado `nombre` con longitud de 40 caracteres, que es la cantidad máxima de caracteres que puede contener esta cadena. En esta longitud máxima se considera un carácter nulo '`\0`', con el cual C finaliza todas las cadenas. Por tanto, si se desea introducir una cadena de hasta 40 caracteres, se debe declarar el arreglo considerando el carácter nulo, es decir,

```
char nombre[41];
```

También, una cadena puede ser inicializada asignándole un literal, por ejemplo:

```
char cadena[] = "abcde";
```

declara una cadena con seis elementos (`cadena[0]` a `cadena[5]`), donde el sexto elemento es el carácter nulo. Si se especifica el tamaño de la cadena de caracteres y la cadena asignada es más larga que el tamaño especificado, se obtendrá un error al momento de la compilación. Por ejemplo:

```
char cadena[4] = "abcde";
```

obtendría un mensaje de error indicando que se excedió los límites del arreglo. Si la cadena asignada es más corta que el tamaño especificado, el resto de los elementos del arreglo se inicializan como nulos.

También se pueden inicializar arreglos de cadenas de caracteres, por ejemplo:

```
char lista[100][60];
```

lo cual crea un arreglo llamado `lista` con 100 filas y cada una de ellas es una cadena de caracteres de longitud máxima de 60.

En el Programa 18 se muestra la declaración de una cadena de caracteres y su posterior uso. Nótese que en la función `scanf`, la variable nombre no necesita estar precedida por el operador `&`, porque el arreglo `nombre` es una dirección.

Programa 18: Declaración de una cadena de caracteres.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    char saludo[] = "Hola ";
6    char nombre[20];
7
8    printf("Introduce tu nombre: ");
9    scanf("%s",nombre);
10   printf("%s%s\n",saludo,nombre);
11   return 0;
12 }
```

Ejemplo 4.2: (5 min)

Escribir y ejecutar en tu sistema el Programa 18.

4.2.1. Funciones `gets` y `puts`

Cuando se introducen caracteres mediante la función `scanf`, una desventaja es que no permite la entrada de varias palabras separadas por espacios en blanco, ya que `scanf` interpreta los espacios como separador de los datos de entrada. Para corroborar esto, ejecute el Programa 18 introduciendo su nombre y apellido separados por un espacio, en la salida sólo se mostrarán los caracteres relacionados al nombre.

Para poder introducir cadenas de caracteres desde el `stdin` que incluyan espacios, se utiliza la función `gets` definida en la biblioteca `stdio.h` Se pueden introducir todos los caracteres del teclado, excepto el carácter de nueva línea '`\n`', que es automáticamente reemplazado por el carácter nulo '`\0`', con el cual C finaliza todas las cadenas. La función `gets` devuelve un puntero[†] al valor leído. Un valor nulo para este puntero indica un error o una condición de fin de archivo (`EOF`, *end of file*) (ver Programa 21).

[†]Variable que almacena la dirección de otra variable.

Cuando se utilizan las funciones `gets`, `getchar` y `scanf` en un mismo programa, puede haber problemas de lectura no deseados cuando se utilizan algunos compiladores. La solución es limpiar el *buffer* asociado al `stdin` mediante la función `fflush` (Windows, en `stdlib.h`) o `fpurge` (Mac Os, en `stdio.h`). Entonces se debe llamar el comando `fflush(stdin)` o `fpurge(stdin)` después de una lectura con la función `scanf` o `getchar`, y antes de una lectura con la función `gets`.

La función `puts` escribe una cadena al `stdout`, y reemplaza el carácter nulo de terminación de la cadena '`\0`' por el carácter de nueva línea '`\n`'. La función `puts` retorna un valor positivo si se ejecuta satisfactoriamente, en caso contrario retorna el valor `EOF`.

En el Programa 19 se muestra el uso de las funciones `gets` y `puts` aplicados al código en el Programa 18.

Programa 19: Funciones `gets` y `puts`.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    char saludo[] = "Hola ";
6    char nombre[50], *pc;
7
8    printf("Introduce tu nombre completo: ");
9    pc = gets(nombre);
10   printf("%s",saludo);
11   puts(pc);
12   return 0;
13 }
```

4.3. Estructuras

Una *estructura* es un tipo de dato que se puede manipular de la misma forma que otros tipos predefinidos como `float`, `int`, `char`, entre otros. Una estructura se puede definir como una colección de datos de diferentes tipos, lógicamente relacionados. En C, una estructura sólo puede contener declaraciones de variables. En otros compiladores, este tipos de construcciones son conocidas como *registros*.

Crear una estructura es definir un nuevo tipo de dato, denominado *tipo estructura*. En la definición del tipo estructura se especifican los elementos que la

componen (*campos*) así como sus tipos. La sintaxis es la siguiente

```
struct tipo_estructura
{
  /* Declaración de los campos */
};
```

donde `tipo_estructura` es el identificador que nombra del nuevo tipo definido. Después de definir un tipo estructura, se puede declarar una variable de este tipo, de la misma manera que se hace con los tipos básicos:

```
struct tipo_estructura variable1, . . . , variableN;
```

Para referirse a un campo específico de la estructura se usa la notación:

```
variable.campo
```

Por ejemplo, se crea un nuevo tipo estructura `ficha`

```
struct ficha
{
  char nombre[40];
  char dirección[80];
  int  edad;
};
```

después se declaran las variables con este nuevo tipo:

```
struct ficha persona1, persona2;
```

donde las variables `persona1` y `persona2` constan de los campos `nombre`, `dirección` y `edad`.

Las declaraciones de las variables `persona1` y `persona2` también se pueden realizar de la siguiente manera:

```
struct ficha
{
  char nombre[40];
  char dirección[80];
  int  edad;
} persona1, persona2;
```

El tipo de dato de los campos sólo puede ser básico, arreglo, estructura, puntero, unión o función. Los campos de una estructura son almacenados secuencialmente, en el mismo orden en que son declarados.

Para declarar un campo como una estructura, es necesario haber declarado previamente ese tipo de estructura, por ejemplo:

```
struct fecha
{
  int día, mes, año;
};

struct ficha
{
  char nombre[40];
  char dirección[40];
  int edad;
  fecha fecha_nacimiento;
};

struct ficha persona;
```

Con una variable declarada como una estructura, pueden realizarse las siguientes operaciones:

- Tomar su dirección por medio del operado `&`.
- Acceder a uno de sus miembros.
- Asignar una estructura a otra con el operador de asignación.

Cuando los elementos de un arreglo son de tipo estructura, el arreglo recibe el nombre de *arreglo de estructuras* o *arreglo de registros*. Por ejemplo, el Programa 20 lee una lista de alumnos y sus notas al final del curso, dando como resultado el porcentaje de alumnos aprobados y reprobados.

Programa 20: Arreglo de estructuras.

```c
#include <stdio.h>
#include <stdlib.h>
#define NA 10

int main(void) {
  struct ficha {
    char nombre[60];
    float  nota;
  } alumnos[NA];
  int n = 0, i, c;
  float aprobados = 0, reprobados = 0;

  while(n < NA) {
    printf("Nombre: ");
    gets(alumnos[n].nombre);
    printf("Nota: ");
    scanf("%f",&alumnos[n++].nota);
    while ((c=fgetc(stdin)) != '\n' && c != EOF);// fflush(stdin)
  }
  for(i=0;i<NA;i++){
    if (alumnos[i].nota > 5)
      aprobados++;
    else
      reprobados++;
  }
  printf("Aprobados: %.2g%%\n",100*aprobados/NA);
  printf("Reprobados: %.2g%%\n",100*reprobados/NA);
  return 0;
}
```

Ejemplo 4.3: (10 min)

Escribir y ejecutar en tu sistema el Programa 20.

5. Punteros

Un *puntero* es una variable que contiene la *dirección de memoria* de un dato o de otra variable que contiene el dato. Esto quiere decir que el puntero apunta al espacio físico donde está el dato o la variable. Un puntero puede apuntar a un objeto de cualquier tipo. Los punteros se utilizan para:

- Crear y manipular estructuras de datos.
- Asignar memoria dinámicamente.
- Proveer el paso por referencia en las llamadas a funciones.

Un puntero se declara de la siguiente manera:

```
tipo_C *var_puntero;
```

donde

- `tipo_C` especifica el tipo de objeto apuntado; puede ser cualquier tipo.
- el carácter '`*`' es el operador unario de *indirección* que significa 'apunta a'.
- `var_puntero` es el nombre de la variable puntero.

Por otro lado, el operador unario de *dirección-de*, representado por el carácter `&`, devuelve como resultado la dirección de su operando. El operador de indirección `*` toma su operando como una dirección y devuelve su contenido. Por ejemplo, en el siguiente código:

```
int a = 10;
int *p;
p = &a;
printf("En la direccion %X está el dato %d\n", p, *p);
```

indica que el apuntador `p` está apuntando a la dirección de la variable `a`, como se muestra en la Figura 5.1, y se imprime algo como:

```
En la dirección E3BF4B18 está el dato 10
```

Es importante resaltar que el tipo de dato del puntero debe ser igual que el de la variable a la que apunta.

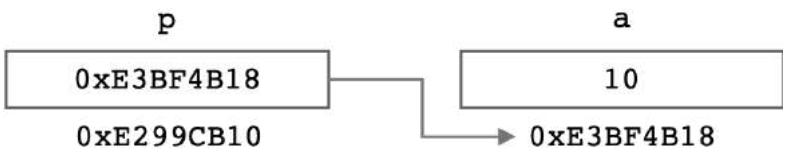


Figura 5.1: El puntero `p` apunta a la dirección de memoria de la variable `a`.

5.1. Operaciones con punteros

Un puntero puede asignarse a otro puntero. Por ejemplo, en el siguiente código:

```
int a = 10, *p, *q;
p = &a;
q = p;
printf("En la direccion %X está el dato %d\n", q, *q);
```

se indica que la dirección que contiene `p` se asigna a `q`, y se imprime algo como:

```
En la dirección E6FD5B18 está el dato 10
```

También, se pueden realizar operaciones aritméticas con punteros. Por ejemplo:

- `b = *pa + 1`, a la variable que está apuntando `pa` se le suma `1` y el resultado se asigna a `b`.
- `b = *(pa + 1)`, el siguiente entero al entero apuntado por `pa` se asigna a `b`.
- `*pb = 0`, a la variable que está apuntando `pb` se le asigna cero.
- `*pb += 2`, a la variable que está apuntando `pb` se le suman dos unidades.
- `(*pb)--`, a la variable que está apuntando `pb` decrementa en una unidad. Sin paréntesis, se decrementaría `pb` y no la variable a la que está apuntando.

Es posible comparar punteros en una expresión de relación. Por ejemplo, se puede comparar un puntero con un valor nulo (`NULL`) para que en el caso `VERDADERO` se detenga el proceso de inserción de cadenas mediante un ciclo, como se muestra en el Programa 21. En la línea `8` se declara el puntero `*fin`, cuyo valor, devuelto por `gets`, se compara con el valor `NULL` para detener el proceso en caso que se hayan oprimido las teclas `Ctrl+Z` (`Ctrl+D` en sistemas basados en Unix).

Programa 21: Comparación de punteros.

```c
#include <stdio.h>
#define FILAS 10
#define COLS 60

int main(void) {
  int n, i=0;
  char *fin; // Valor devuelto por gets
  char lista[FILAS][COLS];

  fputs("Para finalizar pulse Ctrl+Z\n", stdout); // Ctrl+D en
     Unix
  do{
    fputs("Nombre completo: ", stdout);
    fin = fgets(lista[i++],COLS,stdin);
  }while(fin != NULL && i < FILAS);

  if(i < FILAS) i--; // Elimina la ultima entrada Ctrl+Z

  // Escribir la lista en pantalla
  fputs("\nLos nombres son:\n", stdout);
  for(n = 0; n < i; n++)
    fputs(lista[n],stdout);

  return 0;
}
```

Ejemplo 5.1: (10 min)

Escribir y ejecutar en tu sistema el Programa 21.

5.2. Punteros y arreglos

En C, cualquier operación que se pueda realizar mediante indexación de una arreglo, se puede realizar también con punteros. Observe los dos programas que se muestran a continuación. Ambos son funcionalmente idénticos: imprimen en pantalla los valores almacenados en un arreglo. La diferencia entre ambos radica en

cómo se recorre el arreglo. En el caso de la izquierda, se accede a cada elemento del arreglo mediante el incremento de un índice, mientras que en el caso de la derecha, se accede a cada elemento del arreglo mediante el incremento de un apuntador.

```
#include <stdio.h>

int main(void) {
  int lista[]={24,30,15,45};
  int i;

  for (i=0; i<4; i++)
    printf("%d ", lista[i]);

  return 0;
}
```

```
#include <stdio.h>

int main(void) {
  int lista[]={24,30,15,45};
  int i;

  for (i=0; i<4; i++)
    printf("%d ", *(lista+i));

  return 0;
}
```

Esto quiere decir que `lista` es la dirección del comienzo del arreglo. Si esta dirección se incrementa en `1`, o dicho de otra manera si `i` vale `1`, se imprimirá el siguiente número en la lista, esto es, `*(lista+1)` y `lista[1]` representan el mismo valor. Por tanto, hay dos formas para acceder a los elementos de un arreglo:

```
*(arreglo + índice) ó arreglo[índice]
```

Debido a que `*(lista+0)` es igual a `lista[0]`, la asignación `p=&lista[0]` es la misma que la asignación `p=lista`, donde `p` es una variable de tipo puntero. Esto indica que la dirección de comienzo de un arreglo es la misma que la del primer elemento. Por otra parte, después de haber efectuado la asignación `p=lista`, las siguientes expresiones generan resultados idénticos:

```
p[índice], *(p + índice), arreglo[índice], *(arreglo + índice)
```

Nótese que el nombre de un arreglo es una constante y un puntero es una variable; por tanto, operaciones como

```
arreglo = p
arreglo++
```

son incorrectas, mientras que son correctas operaciones como

```
p = arreglo
p++
```

Cabe señalar que la teoría expuesta anteriormente es perfectamente aplicable a cadenas de caracteres, ya que se trata de arreglos de caracteres. La forma de declarar un puntero a una cadena de caracteres es

```
char *cadena;
```

Por ejemplo:

```
char *nombre = "Javier Hernandez";
printf("%s", nombre);
```

se asigna a `nombre` la dirección de comienzo de la cadena de caracteres especificada. El compilador añade al final de la cadena el carácter nulo, para detectar en operaciones posteriores el final de la misma. En el Programa 22 se muestra la forma de calcular la longitud de una cadena de caracteres utilizando punteros.

Programa 22: Determina el largo de una cadena con puntero.

```
1  #include <stdio.h>
2
3  int largo(char *);
4
5  int main(void)
6  {
7    char *cadena = "hola, mundo";
8
9    printf("\'%s\' tiene %d caracteres\n", cadena, largo(cadena));
10
11   return 0;
12 }
13
14 /* Funcion que devuelve el largo de la cadena */
15 int largo(char *str)
16 {
17   char *p = str;
18   while(*p != '\0')
19     p++;
20   return((int)(p-str));
21 }
```

El ciclo formado por la sentencia `while` (líneas `16` y `17`) realiza las siguientes operaciones:

1. Toma el carácter contenido en la dirección dada por `p`.
2. Verifica si es un carácter nulo. Si es así, el ciclo termina.
3. Si no es un carácter nulo, pasa a la siguiente dirección y regresa al Punto 1.

En la línea `18`, la diferencia `p-str` provee la longitud de la cadena, es decir, la cantidad de direcciones que recorrió desde la primera dirección.

Ejemplo 5.2: (10 min)

Modificar el Programa 22 para que se ingrese una cadena de caracteres por teclado usando la función `gets` y determine el largo de esa cadena.

El Programa 23 copia una cadena de caracteres en otra usando punteros.

Programa 23: Copia una cadena de caracteres en otra.

```
1  #include <stdio.h>
2  #define N 80
3
4  void copiar(char *, char *);
5
6  int main(void)  {
7    char cadena1[N], cadena2[N];
8
9    printf("Introduce una cadena: ");
10   fgets(cadena1, N, stdin);
11
12   copiar(cadena2,cadena1);
13   printf("La cadena copiada es %s \n",cadena2);
14
15   return 0;
16 }
17
18 void copiar(char *p, char *q)
19 {
20   while((*p++ = *q++) != '\0');
21 }
```

El ciclo formado por la sentencia `while` (línea `17`) realiza las siguientes operaciones:

1. Toma el carácter contenido en la dirección dada por `q` y lo copia en la direc-

ción dada por `p`.

2. Verifica si el carácter de `p` es el carácter nulo. Si es así termina el ciclo.
3. Si `p` no es un carácter nulo, pasa a apuntar a la siguiente dirección y lo mismo sucede con `q`. El proceso se repite desde el Punto 1.

Ejemplo 5.3: (5 min)

Escribir y ejecutar en tu sistema el Programa 23.

6. Funciones

Una *función* es una colección independiente de declaraciones y sentencias, generalmente enfocadas a realizar una tarea específica. Todo programa en C consta de al menos una función, la función `main`, que es de donde comienza la ejecución del programa. Además de ésta, pueden haber otras funciones cuya finalidad es descomponer el problema general en subproblemas más fáciles de resolver.

Cuando se llama a una función, el control se pasa a la misma para su ejecución; y cuando finaliza, el control es devuelto a la función que la llamó para continuar con la ejecución a partir de la sentencia que efectuó la llamada, como se muestra en la Figura 6.1.

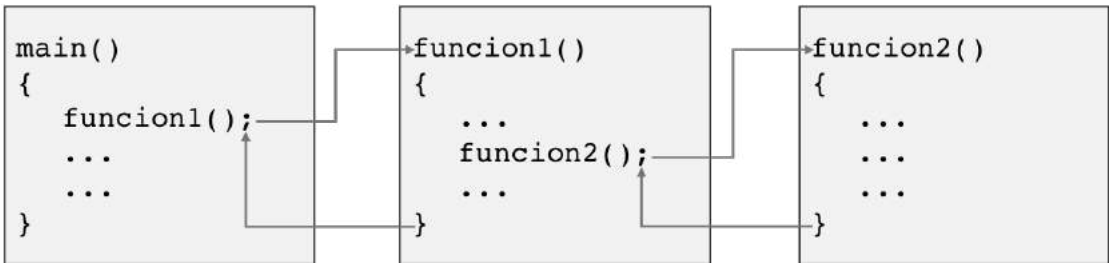


Figura 6.1: Llamadas a funciones en C.

6.1. Definición de una función

La definición de una función consta de la *cabecera de la función* y del *cuerpo de la función*. La sintaxis correspondiente es:

```
clase tipo identificador(tipo id1, . . . , tipo idN)
{
   /* cuerpo de la función */
   declaraciones;
   sentencias;
}
```

donde

- `clase` define el ámbito de la función, esto es, desde donde puede ser llamada. La clase puede ser `static` o `extern`. Una función `static` es visible solamente

en el archivo fuente en el cual está definida, mientras que una función `extern` es visible para todos los archivos fuente que componen el programa. Si se omite la clase, por defecto la función es `extern`.

- `tipo` indica el tipo de valor devuelto por la función. Puede ser de cualquier tipo básico o tipo definido por el usuario. Por defecto, el tipo es `int`. Una función no puede retornar un arreglo o función, aunque si puede retornar un puntero a un arreglo o a una función.
- `identificador` es el nombre de la función. Si el nombre va precedido por el operador de indirección (`*`), el valor devuelto por la función es un puntero.
- `parámetros_formales` componen la lista de argumentos de la función. Esta lista consta de un conjunto de variables con sus tipos separados por comas y encerrados entre paréntesis. Los parámetros formales son variables que reciben los valores que se pasan en la llamada a la función. Si no se pasan argumentos a la función, la lista de parámetros formales puede ser sustituida por la palabra clave `void`.

El cuerpo de una función esta formado por las sentencias que ejecuta la función. También pueden contener declaraciones de variables utilizadas en dichas sentencias, las cuales son, por defecto, locales en la función.

La sentencia `return` devuelve un valor del tipo especificado en la cabecera de la función, retornando de esta manera el control a la función que hizo la llamada. La sintaxis es

```
return (expresión);
```

Si en la sentencia `return` no se especifica una `expresión`, la función no devuelve un valor.

La declaración de una función, denominada también *función prototipo*, permite conocer el nombre, el tipo del resultado, el tipo de los parámetros formales y opcionalmente sus nombres. No define el cuerpo de la función. Esta información permite al compilador verificar los tipos de los parámetros actuales por cada llamada a la función. Una función prototipo tiene la misma sintaxis que la definición de una función, excepto que ésta termina con un punto y coma. Una función no puede ser llamada si previamente no está definida o declarada. En caso de que una función haya sido definida antes del bloque de la función `main`, entonces no es necesario declarar su función prototipo, en caso contrario, se declara su función

prototipo antes de la función `main`. Observe los dos programas que se muestran a continuación. Ambos son funcionalmente idénticos: dos valores introducidos por teclado se suman en la llamada a la función `suma`. La diferencia entre ambos radica en que el programa de la izquierda declara una función prototipo debido a que la definición de la función `suma` es posterior a su llamada en la función `main`, mientras que en el programa de la derecha, la definición de la función `suma` es previo a su llamada, por lo que no requiere declarar una función prototipo.

```
#include <stdio.h>

/* Función prototipo */
float suma(float, float);

int main(void){
  float a, b, c;

  printf("Introduce a y b: ");
  scanf("%f %f", &a, &b);

   /* Llamada a la función */
  c = suma(a, b);

  printf("%.2f+%.2f=%.2f\n", a, b, c);

  return 0;
}

/* Definición de la función */
float suma(float a, float b)
{
  return a+b;
}
```

```
#include <stdio.h>

/* Definición de la función */
float suma(float a, float b)
{
  return a+b;
}

int main(void){
  float a, b, c;

  printf("Introduce a y b: ");
  scanf("%f %f", &a, &b);

   /* Llamada a la función */
  c = suma(a, b);

  printf("%.2f+%.2f=%.2f\n", a, b, c);

  return 0;
}
```

6.2. Paso de parámetros por valor y referencia

Cuando se llama una función, hay dos formas de pasar los parámetros actuales a sus correspondientes parámetros formales:

1. **Paso de parámetros por valor:** cuando el control pasa a la función, los valores actuales de los parámetros se copian automáticamente en sus co-

rrespondientes parámetros formales, de modo que al regresar el control a la función que la llamó, los parámetros conservan sus valores originales. Pueden ser transferidos constantes, variables y expresiones.

2. **Paso de parámetros por referencia:** los parámetros transferidos a la función no son valores sino las direcciones de las variables que contienen esos valores, de manera que los parámetros actuales pueden sufrir cambios al regresar el control a la función que la llamó. Esto es útil si se desea devolver más de un parámetro al mismo tiempo. Pueden ser transferidos variables de cualquier tipo de datos, así como arreglos y funciones.

En los códigos anteriores, la forma en que se han pasado los parámetros a la función `suma` fue por valor. Para pasar parámetros por referencia, la definición de la función lleva la siguiente sintaxis:

```
clase tipo identificador(tipo *id1, tipo id2, . . . , tipo *idN)
{
   /* cuerpo de la función */
   declaraciones;
   sentencias;
}
```

donde los parámetros formales pasados por referencia van precedidos del operador de indirección (`*`). Para hacer una llamada a una función a la que se le pasarán parámetros por referencia, se utiliza el operador dirección-de (`&`) antes del nombre de la variable, lo que indica que se le está transfiriendo la dirección del parámetro actual a su correspondiente parámetro formal, el cual tiene que ser un puntero. Para pasar la dirección de un arreglo, no es necesario el operador `&` antes de su nombre, ya que el nombre del arreglo es un puntero a dicho arreglo.

Observe los dos programas que se muestran a continuación. Ambos son funcionalmente idénticos: dos valores introducidos por teclado se suman en la llamada a la función `suma`. La diferencia entre ambos radica en que en el programa de la izquierda, el resultado de la suma se asigna a la variable `c` al retornar de la función `suma`, mientras que en el programa la derecha, el resultado de la suma se almacena en la variable `c` cuya dirección fue pasada por referencia.

```c
#include <stdio.h>

/* Función prototipo */
float suma(float, float);

int main(void){
  float a, b, c;

  printf("Introduce a y b: ");
  scanf("%f %f", &a, &b);

   /* Llamada a la función */
  c = suma(a, b);

  printf("%.2f+%.2f=%.2f\n", a, b, c);

  return 0;
}

/* Definición de la función */
float suma(float a, float b)
{
  return a+b;
}
```

```c
#include <stdio.h>

/* Función prototipo */
void suma(float, float, float *);

int main(void){
  float a, b, c;

  printf("Introduce a y b: ");
  scanf("%f %f", &a, &b);

   /* Llamada a la función */
  suma(a, b, &c);

  printf("%.2f+%.2f=%.2f\n", a, b, c);

  return 0;
}

/* Definición de la función */
void suma(float a, float b, float *c)
{
  *c = a+b;
}
```

Para pasar todos los elementos de un arreglo a una función, se pone en la lista de parámetros actuales el nombre del arreglo, que es la dirección de comienzo del arreglo, y en la lista de parámetros formales el nombre del arreglo seguido de sus dimensiones. Esto se ejemplifica en el Programa 24 que muestra un código para pasar por referencia una matriz a una función llamada `LlenaMatriz`, en donde cada elemento de la matriz se almacena el valor del producto de sus índices. La matriz se inicializa con valores nulos en la línea `9`, se llena la matriz en la línea `11` mediante la función `LlenaMatriz`, y en la línea `16` se imprime en pantalla la matriz resultante con valores no nulos.

Programa 24: Paso por referencia de una matriz.

```c
#include <stdio.h>
#define FILAS 5
#define COLS 5

void LlenaMatriz(int[FILAS][COLS]);

int main(void){

  int i, j, matriz[FILAS][COLS] = {};

  LlenaMatriz(matriz);

  for (i=0; i<FILAS; i++){
    printf("\n");
    for (j=0; j<COLS; j++){
      printf("%d\t",matriz[i][j]);
    }
  }
  return 0;
}

void LlenaMatriz(int mat[FILAS][COLS]){
  int c, f;
  for (f=0; f<FILAS; f++)
    for (c=0; c<COLS; c++)
      mat[f][c] = f*c;
}
```

Ejemplo 6.1: (5 min)

Escribir y ejecutar en tu sistema el Programa 24.

Referencias

[1] B. W. Kernighan & D. M. Ritchie. *The C Programming Language*. 2nd Edition. Prentice Hall, 1988.

[2] K. N. King. *C Programming: A Modern Approach*. 2nd Edition. W. W. Norton & Company, 2008.

[3] Greg Perry. *Absolute Beginner's Guide To C*. 2nd Edition. Sams Publishing, 1994.

[4] Stephen Prata. *C Primer Plus*. 5th Edition. Sams, 2004.

[5] Peter V. Linden. *Expert C Programming: Deep C Secrets*. Prentice Hall, 1994.

[6] Francisco Javier Ceballos. *Enciclopedia del Lenguaje C*. Alfaomega, 1997.