# Introducción a la Programación

**Notas de clase**

**para el curso**

# Programación I

Licenciatura y Profesorado en Ciencias de la Computación

Ingeniería en Informática

Ingeniería en Computación

Aristides Dasso

Ana Funes

Área de Programación y Metodologías de Desarrollo del Software
Departamento de Informática
Facultad de Ciencias Físico-Matemáticas y Naturales
Universidad Nacional de San Luis
Argentina
2014

# TABLA DE CONTENIDO

**PRÓLOGO** ..... 1

**CONSIDERACIONES GENERALES** ..... 1

**1 LAS COMPUTADORAS** ..... 2

**1.1 INTRODUCCIÓN** ..... 2
**1.2 LAS PARTES DE UNA COMPUTADORA** ..... 3
INTRODUCCIÓN ..... 3
LAS PARTES INTERNAS ..... 3
Unidad Central de Proceso ..... 3
Memoria principal ..... 5
Unidades de medida de almacenamiento ..... 6
Tecnologías de fabricación ..... 7
LAS PARTES EXTERNAS ..... 7
Teclado ..... 7
Video ..... 8
Mouse (Ratón) ..... 9
Memoria auxiliar ..... 9
Impresora ..... 11
**1.3 LA INFORMACIÓN DENTRO DE LA COMPUTADORA** ..... 12
**1.4 EL FUNCIONAMIENTO** ..... 14
**1.5 EL SISTEMA OPERATIVO** ..... 16

**2 PROGRAMAR** ..... 18

**2.1 INTRODUCCIÓN** ..... 18
**2.2 PROGRAMACIÓN DE COMPUTADORAS** ..... 20
**2.3 PROGRAMACIÓN Y RESOLUCIÓN DE PROBLEMAS** ..... 21
**2.4 EL PROCESO DE RESOLUCIÓN DE PROBLEMAS** ..... 22
INTRODUCCIÓN ..... 22
ETAPAS DE LA RESOLUCIÓN DE PROBLEMAS ..... 23
**2.5 LENGUAJES DE PROGRAMACIÓN: LENGUAJES DE MÁQUINA, LENGUAJES ENSAMBLADORES Y LENGUAJES DE ALTO NIVEL** ..... 25
**2.6 LOS PARADIGMAS DE LA PROGRAMACIÓN** ..... 26
**2.7 EL PARADIGMA DE PROGRAMACIÓN IMPERATIVA** ..... 27
**2.8 ETAPAS EN LA CONSTRUCCIÓN DE UN PROGRAMA** ..... 27

**3 PROGRAMACIÓN EN C ..... 29**

**3.1 INTRODUCCIÓN ..... 29**
**3.2 LA ESTRUCTURA GENERAL DE UN PROGRAMA EN C ..... 29**
**3.3 PROGRAMACIÓN Y DATOS ..... 30**
**3.4 TIPOS DE DATOS, OPERADORES Y EXPRESIONES EN C ..... 32**
TIPOS DE DATOS ARITMÉTICOS ..... 33
Enteros ..... 33
Caracteres ..... 33
Flotantes o Reales ..... 34
OPERADORES Y EXPRESIONES ..... 35
Operadores y expresiones aritméticas ..... 35
Operadores y expresiones relacionales ..... 36
Operadores y expresiones lógicas ..... 37
Operador de asignación ..... 37
Conversiones de tipos – Operador cast ..... 38
Operadores de incremento y decremento ..... 40
**3.5 ESTRUCTURAS DE CONTROL ..... 40**
SECUENCIA ..... 41
SELECCIÓN ..... 42
ITERACIÓN ..... 45

**4 MODULARIDAD ..... 49**

**4.1 LAS FUNCIONES EN C ..... 49**
DEFINICIÓN DE UNA FUNCIÓN EN C ..... 51
INVOCACIÓN DE UNA FUNCIÓN ..... 51
PASAJE DE PARÁMETROS ..... 51
**4.2 ALCANCE O ÁMBITO DE LOS IDENTIFICADORES ..... 53**
**4.3 TIEMPO DE VIDA DE LOS IDENTIFICADORES ..... 54**

**5 PUNTEROS ..... 57**

**5.1 PUNTEROS EN C ..... 57**
PUNTEROS A **`VOID`** ..... 59
**5.2 PASAJE DE PARÁMETROS POR REFERENCIA O DIRECCIÓN EN C ..... 60**
**5.3 ASIGNACIÓN DINÁMICA DE LA MEMORIA ..... 63**

**6 ESTRUCTURAS DE DATOS ..... 65**

**6.1 INTRODUCCIÓN ..... 65**
**6.2 CAPACIDAD DE LAS ESTRUCTURAS DE DATOS ..... 66**
CAPACIDAD DINÁMICA ..... 66
CAPACIDAD ESTÁTICA ..... 66
**6.3 OPERACIONES ..... 66**
**6.4 ORDEN DE LOS ELEMENTOS ..... 67**
ORDEN CRONOLÓGICO ..... 67
ORDEN NO CRONOLÓGICO ..... 68
**6.5 SELECTOR DE LOS ELEMENTOS ..... 68**
**6.6 TIPO BASE ..... 68**

**7 ESTRUCTURAS DE DATOS: ARREGLOS Y REGISTROS ..... 69**

**7.1 ARREGLOS ..... 69**
ORDEN CRONOLÓGICO ..... 69
CAPACIDAD ..... 69
OPERACIONES ..... 70
REPRESENTACIÓN GRÁFICA ..... 70
**7.2 LOS ARREGLOS EN C ..... 70**
DECLARACIONES DE ARREGLOS ..... 70
EJEMPLO DE USO ..... 71
ARREGLOS MULTIDIMENSIONALES ..... 72
RELACIÓN ENTRE ARREGLOS Y PUNTEROS EN C ..... 73
ARITMÉTICA DE PUNTEROS ..... 74
PASAJE DE UN ARREGLO COMO PARÁMETRO DE UNA FUNCIÓN ..... 75
ARREGLOS DE CARACTERES Y STRINGS ..... 78
STRINGS, ARREGLOS DE CARACTERES Y PUNTEROS A **CHAR** ..... 80
**7.3 REGISTROS ..... 80**
ORDEN CRONOLÓGICO ..... 81
CAPACIDAD ..... 81
OPERACIONES ..... 81
REPRESENTACIÓN GRÁFICA ..... 81
**7.4 LOS REGISTROS EN C: STRUCTS ..... 81**
DECLARACIÓN DE STRUCTS ..... 81
OPERACIONES CON STRUCTS ..... 83
INICIALIZACIÓN DE UN STRUCT ..... 83
ACCESO A LOS CAMPOS DE UN STRUCT ..... 83
PASAJE DE UN REGISTRO COMO PARÁMETRO DE UNA FUNCIÓN ..... 83
USO DE **TYPEDEF** ..... 85
PUNTEROS A STRUCTS ..... 86

**8 ESTRUCTURAS DE DATOS: PILAS Y FILAS ..... 87**

**8.1 PILAS ..... 87**
ORDEN CRONOLÓGICO ..... 87
CAPACIDAD ..... 87
OPERACIONES ..... 87
REPRESENTACIÓN GRÁFICA ..... 88
**8.2 FILAS O COLAS ..... 88**
ORDEN CRONOLÓGICO ..... 88
CAPACIDAD ..... 89
OPERACIONES ..... 89
REPRESENTACIÓN GRÁFICA ..... 89

**9 ESTRUCTURAS DE DATOS: LISTAS UNI Y BIDIRECCIONALES ..... 90**

**9.1 LISTAS UNIDIRECCIONALES ..... 90**
COMPOSICIÓN DE LOS ELEMENTOS ..... 90
ORDEN CRONOLÓGICO ..... 90
CAPACIDAD ..... 90
OPERACIONES ..... 90
REPRESENTACIÓN GRÁFICA ..... 91
**9.2 LISTAS BIDIRECCIONALES ..... 91**
COMPOSICIÓN DE LOS ELEMENTOS ..... 91
ORDEN CRONOLÓGICO ..... 91
CAPACIDAD ..... 92
OPERACIONES ..... 92
REPRESENTACIÓN GRÁFICA ..... 92
**9.3 GENERALIDAD DE LAS LISTAS ..... 93**

**10 ESTRUCTURAS DE DATOS: ESTRUCTURAS MULTINIVEL ..... 95**

**10.1 PARTICULARIDADES DE LAS ESTRUCTURAS DE MULTINIVEL ..... 96**
**10.2 ESTRUCTURAS ESTÁTICAS Y DINÁMICAS ..... 96**
**10.3 OPERACIONES ..... 96**

**11 IMPLEMENTACIÓN DE ESTRUCTURAS ..... 97**

**11.1 INTRODUCCIÓN ..... 97**
**11.2 ADMINISTRACIÓN DE LOS ESPACIOS LIBRES ..... 97**
DESBORDE Y DESFONDE ..... 98

MÉTODOS DE ADMINISTRACIÓN DE LOS ESPACIOS LIBRES ..... 98
Administración estática ..... 98
Administración dinámica ..... 99
**11.3 IMPLEMENTACIÓN DE PILAS ..... 99**
CON DESPLAZAMIENTO ..... 99
Con variable para la base ..... 100
Con una marca ..... 100
SIN DESPLAZAMIENTO ..... 101
**11.4 IMPLEMENTACIÓN DE FILAS O COLAS ..... 102**
CON DESPLAZAMIENTO ..... 103
Con variable para el último ..... 103
Con una marca ..... 104
SIN DESPLAZAMIENTO ..... 106
**11.5 IMPLEMENTACIÓN DE LISTAS ..... 109**
CON DESPLAZAMIENTO ..... 109
Con variable para el último ..... 109
Con una marca ..... 111
SIN DESPLAZAMIENTO ..... 113
Con arreglos separados ..... 113
Con un solo arreglo ..... 117

**12 RECURSIVIDAD ..... 119**

**12.1 NOCIONES GENERALES ..... 119**
**12.2 MÓDULOS RECURSIVOS ..... 119**
**12.3 RECURSIVIDAD. EJEMPLOS ..... 121**
SUCESIÓN DE FIBONACCI ..... 121
"TORRES DE HANOI" ..... 122
**12.4 TIPOS DE RECURSIVIDAD ..... 125**
RECURSIVIDAD LINEAL Y RECURSIVIDAD MÚLTIPLE ..... 125
RECURSIVIDAD DIRECTA E INDIRECTA ..... 125
**12.5 PROFUNDIDAD DE LA RECURSIÓN ..... 126**
**12.6 DATOS RECURSIVOS ..... 127**
**12.7 IMPLEMENTACIÓN DE LISTAS CON DATOS RECURSIVOS ..... 129**
**12.8 CASOS PARTICULARES DE RECURSIÓN ..... 130**
INICIALIZACIÓN DE LA RECURSIÓN ..... 130
ACTUALIZACIÓN DE VALORES ENTRE LLAMADAS ..... 132
REPETICIONES ANIDADAS ..... 132
**12.9 ITERACIÓN Y RECURSIÓN ..... 133**
**12.10 VENTAJAS E INCONVENIENTES DE LA RECURSIÓN ..... 134**

**APÉNDICE A LÓGICA ..... 135**

**A.1 INTRODUCCIÓN ..... 135**
**A.2 PROPOSICIONES ..... 135**
**A.3 OPERADORES, CONECTIVAS O FUNTORES ..... 137**
CONJUNCIÓN ..... 138
DISYUNCIÓN ..... 138
CONDICIONAL O IMPLICACIÓN ..... 139
Condición suficiente y condición necesaria ..... 140
BICONDICIONAL O EQUIVALENCIA ..... 140
NEGACIÓN ..... 141
**A.4 EQUIVALENCIAS LÓGICAS ..... 141**

**APÉNDICE B TABLA ASCII (AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE) ..... 143**

# PRÓLOGO

Antes de comenzar a usar estas notas de clase nos ha parecido una buena idea empezar con algunas consideraciones generales sobre la o las materias para las que están dedicadas, y sobre todo para dar algunas aclaraciones e ideas para guiar al alumno en su tarea de aprendizaje.

¿Cuál es el objetivo? Introducir a la programación de las computadoras. Para quienes hagan Ciencias de la Computación esto es la introducción a uno de los aspectos más importantes de la carrera.

Algunas veces parece conveniente dar algunos consejos sobre cómo encarar el estudio y la lectura de los textos empleados. En la sección *Consideraciones generales* se hace eso. Esto no es una regla a seguir obligatoriamente. Son solo algunas ideas que a algunos nos han resultado útiles. En última instancia el que pone el mayor esfuerzo en la tarea de enseñanza y aprendizaje es quien aprende. De nada vale asistir a clases si se tiene la mente puesta en otra cosa. De nada vale llevar los apuntes y libros bajo el brazo de un lado a otro. ¡No se aprende por *ósmosis sobacal*! De nada vale pedir que un compañero o un docente resuelva los ejercicios del práctico por nosotros. Estos están para reafirmar los conocimientos a través de la práctica: haciendo también se aprende. Más vale un ejercicio trabajado que diez resueltos por un compañero, aunque se haya estado al lado del mismo mientras este lo hacía.

## CONSIDERACIONES GENERALES

Los textos de temas formales, no reiterativos y sintéticos, cuando se explican en términos no formales resultan reiterativos y poco sintéticos; y algunas veces hasta ambiguos y fácilmente pueden inducir a confusión. Generalmente su explicación está en su formalidad: si se conoce el lenguaje todo está ahí, no hay necesidad de exégesis.

Su gradualidad (de menos complicado a más complicado) hace necesario un conocimiento previo, donde muchos términos, notaciones y conceptos que ya han sido explicados no se necesitan volver a explicar (no hay reiteración), sólo hay que acordarse o sino volver atrás y repasarlos.

Generalmente, se entiende que estos textos no necesitan de una guía de lectura. El texto en sí mismo debería (¿debería?) ser suficiente guía.

Como ya dijimos, las siguientes son unas ideas indicativas de cómo leer un texto dado y no son obligatorios de ninguna manera. Cada uno tiene su forma de leer, de estudiar, un texto.

Así, con ese espíritu, veamos algunas ideas al respecto:

- Lea todo el texto una primera vez.
- Si hay alguna palabra que no entendió, búsquela en el diccionario. ¿El significado, que encontró en el diccionario, para las palabras que no entendió, tiene sentido en la frase del texto?
- Recuerde que existe una jerga en muchos temas, especialmente de Computación. ¿Está seguro que alguna de las palabras que no sabía su significado no es parte de la jerga de Computación? Si tiene dudas al respecto consulte a los docentes.
- Lo que no entendió ¿no será porque hay un tema anterior que no recuerda bien? Repase lo anterior si cree que es necesario. Repáselo ante la menor duda, nunca está de más.
- No crea que con leerlo una sola vez es suficiente. En realidad hay que estudiarlo.
- Estudiar el texto significa leerlo, analizarlo, trabajarlo tantas veces como sea necesario hasta comprender el tema.

# 1 LAS COMPUTADORAS

## 1.1 INTRODUCCIÓN

¿Qué es una computadora? La respuesta más obvia y directa a esta pregunta es: una computadora es una máquina; que es, por otro lado, correcta.

Es cierto que las computadoras son máquinas, tal como las lavarropas, los automóviles, los receptores de radio o televisión, etc. Son sin embargo, por el momento al menos, las máquinas más versátiles y complejas en su funcionamiento potencial que ha construido la humanidad.

¿Por qué hablamos de funcionamiento potencial? Porque como toda máquina, las computadoras necesitan que se las haga funcionar. Sin embargo, las máquinas, en general, tienen una función definida que realizan, ejecutan, cuando se las pone en funcionamiento: un lavarropas lava ropa, un automóvil se mueve, un receptor (ya sea de radio o televisión) recepciona.

Las computadoras, por otro lado, cuando se las hace funcionar *computan*, pero esta acción, en las computadoras, es más compleja, más amplia, más versátil que lo que puede pensarse inicialmente. Justamente por la capacidad de realizar un número muy grande y diverso de acciones (algunas de las cuales son muy complicadas en sí mismas) es que las computadoras pueden llegar a ser tan complejas.

Es que las acciones básicas con que han sido diseñadas las computadoras son mucho más generales que las de cualquier otra máquina. En efecto, cualquier máquina tiene un conjunto de acciones básicas, digamos las instrucciones que la máquina "entiende", y que le permiten actuar. Así un automóvil puede frenar, acelerar, doblar, etc. Las acciones básicas, de ésta máquina tan difundida, están orientadas hacia la tarea para la cual ha sido diseñado el automóvil: transportar algo o alguien de un punto a otro. En el caso de las computadoras las acciones básicas son mucho más generales y se eligen justamente para permitir que el conjunto de ellas sea lo suficientemente amplio, por lo menos en su alcance, para que la computadora pueda ser utilizada en actividades muy distintas unas de otras.

El objetivo básico fundamental de una computadora es *ejecutar* programas, que pretenden resolver problemas.

Justamente, por esta razón, es más importante la *programación* de las computadoras que la de otras máquinas. Claro es que uno puede programar un viaje, un traslado de un punto a otro, en automóvil: cuándo acelerar, cuándo frenar, cuándo doblar, todo esto sin contar los imprevistos, por supuesto, para los que será necesario programar también. Es decir, algunas máquinas pueden programarse. De hecho hay más de una máquina que se pueden programar, como por ejemplo los hornos de microondas, los lavarropas, los lavaplatos, los GPS, etc. Algunas de ellas tienen ya unos cuantos años, como los telares del siglo XVII en los que se programaban los dibujos de las telas.

Sin embargo, son estas nuevas máquinas, las computadoras, tan versátiles, y que se aplican a la solución de tantos y diversos problemas, las que han llevado la programación a constituir una destacada actividad por el volumen que la misma implica. Es que, más que con ninguna otra máquina, la programación es, de lejos, la actividad fundamental para el funcionamiento de ellas.

El funcionamiento de las computadoras más comunes, está basado, por lo menos por el momento, en componentes (transistores, resistencias, etc.) con los cuales se construyen circuitos electrónicos. Algunos de estos se encuentran dispuestos (o empaquetados si se quiere) en "pastillas", que se conocen con el nombre familiar de *chips*, generalmente con forma de rectángulo o cuadrado, de un tamaño que oscila en los 3 $cm^2$ o aún menor, achicándose año a año. Estos tienen un número variable (desde 4 a más de 200) de conectores que permiten conectar los circuitos que están dentro de la pastilla con otros circuitos y elementos electrónicos (otras pastillas, por ejemplo). A estas pastillas se las llama *circuitos integrados* (CI), o en inglés *integrated circuits* (IC). Estos circuitos integrados también aumentan, año a año, la densidad (cantidad) de componentes básicos que se integran en ellos.

Los CI se encuentran instalados sobre superficies planas llamadas *plaquetas* o *tarjetas*, (en inglés *cards* o

*boards*) que permiten la interconexión de los CI y otros elementos (capacitores, resistencias, etc.) electrónicos. Como estas conexiones se encuentran dispuestas sobre la superficie de las plaquetas o tarjetas (impresas, grabadas, etc.) también se las llama tarjetas impresas, en inglés *printed circuit boards*.

Las tarjetas impresas se conectan a otras tarjetas impresas por medio de gran cantidad de conectores, (50 o más), que permiten la interconexión de varias tarjetas y que se conocen con el nombre de bus, porque permiten canalizar muchos impulsos eléctricos de unos elementos electrónicos a otros al mismo tiempo.

Las computadoras se construyen, en definitiva, combinando juiciosamente distintos tipos de circuitos integrados sobre tarjetas impresas y estas, a su vez, entre sí, generalmente por medio de uno (o más) buses.

## 1.2 LAS PARTES DE UNA COMPUTADORA

### INTRODUCCIÓN

A las computadoras, por su amplia utilización así como por su aplicación a la solución de tantos y diversos problemas, se tiene tendencia a antropomorfizarlas. Es decir, atribuirles características propias de los seres humanos, particularmente la capacidad de "pensar". Sin embargo, ya hemos dicho que las computadoras no son más que máquinas y si bien esta comparación con los seres humanos es útil para comprender algunos detalles de su funcionamiento no es conveniente abusar de ella.

Así, al hablar de las partes de la computadora y usando (¿abusando?) de esta analogía humanoide:

Hablaremos de los sentidos (ver, escuchar, olfatear, gustar, oír, etc.) y diremos que son las *entradas* a la computadora, por donde ésta, a través de los órganos de los sentidos (ojos, oídos, nariz, etc.), capta el medio ambiente.

Hablaremos de las acciones (hablar, escribir, andar, etc.) y diremos que son las *salidas* de la computadora, y que, por medio de los órganos adecuados (cuerdas vocales, manos, piernas, etc.), acciona sobre el mundo exterior.

Por supuesto que las computadoras no tienen los sentidos, ni las acciones, ni los órganos correspondientes de la misma forma que los seres humanos. Pero a los efectos de una rápida explicación estas analogías son suficientes.

En general, es posible separar (arbitrariamente y al solo efecto de brindar una explicación superficial de las mismas) las partes de una computadora en dos: *internas* y *externas*. La primera tiene que ver con las *actividades internas* que pueda realizar la máquina y la segunda con los *componentes de la computadora* dedicados a recibir impulsos del medio ambiente y accionar sobre él.

Podemos decir que la parte interna es el "cerebro" de las computadoras, mientras que las partes externas tienen que ver con los "sentidos" de la máquina así como con los medios que utilice para accionar sobre el mundo exterior: diríamos que es el equivalente de sus "manos", "voz", etc. Muchas veces se habla de *periféricos* para referirse a las partes externas de la computadora, quizás porque en el pasado las primeras computadoras tenían lo que hemos llamado partes externas, en armarios alrededor, en la periferia, de la máquina principal.

### LAS PARTES INTERNAS

Decíamos que la parte interna de las computadoras podía compararse con el "cerebro" de las mismas. Podemos distinguir aquí dos partes: la *Unidad Central de Proceso* y *la memoria principal*.

#### UNIDAD CENTRAL DE PROCESO

La *Unidad Central de Proceso* (UCP) o *Central Processing Unit* (CPU) en inglés, está compuesta de los

circuitos electrónicos destinados a ordenar el funcionamiento general de la máquina, así como de reconocer un número fijo y determinado de órdenes o *instrucciones* que son las únicas acciones que la computadora puede realizar y sobre las que se construyen (a través de programas que se hacen empleando dichas instrucciones) las acciones que la computadora realiza.

Estas instrucciones se conocen generalmente bajo el nombre de *conjunto o juego de instrucciones*, o *instruction set* en inglés. En efecto, uno de los aspectos más importantes de una CPU es que la misma sólo reconoce un conjunto de instrucciones y sólo esas. Es empleando juiciosamente estas instrucciones que pueden hacerse programas para que una computadora realice las tareas que se pretende de ella.

Este conjunto de instrucciones consta, generalmente, de 50 a 200 instrucciones, aún cuando haya CPUs que solo reconozcan una instrucción y otras más de 200.

Quiere decir que las instrucciones, que una CPU reconoce, son las que han sido incorporadas, a través de circuitos electrónicos (por lo menos en la mayoría de las computadoras en uso corriente), por el fabricante de la misma.

El criterio que cada fabricante tiene para decidir qué instrucciones incorporar a la CPU que él fabrica es esencialmente comercial. En efecto, incorporará aquellas instrucciones que, luego de un análisis de mercado y otros, considere que hará más atractiva su CPU en el mercado y le permitirá venderla más fácilmente por sobre la de sus competidores.

Está claro que una CPU que reconoce cincuenta instrucciones, por ejemplo, no necesariamente ejecuta programas (hechos con esas instrucciones) de cincuenta instrucciones. De la misma manera que un ser humano que conoce sólo tres mil palabras puede leer libros que contienen muchas más de tres mil palabras ya que puede leer y entender cada palabra varias veces, una CPU puede ejecutar programas muy largos, siempre y cuando estén formados de instrucciones que ella reconoce.

La CPU será, entonces, la encargada de *ejecutar* las instrucciones que se le den ordenadas en programas. Estas instrucciones serán ejecutadas en un *orden secuencial*, es decir, una después de la otra y a un ritmo o velocidad predeterminada. Este ritmo, esta velocidad, que permite a cada instrucción ejecutarse aún cuando no todas ellas tomen el mismo tiempo para hacerlo, está controlada por circuitos especiales dentro de la CPU, llamados generalmente *Unidad de Control*. Estos circuitos emplean un *reloj* y una *tabla* especial que les permite conocer cuánto tiempo tarda cada instrucción en ejecutarse.

Este reloj produce *ticks*, *ciclos* o *pulsos* a un ritmo determinado, es decir que hay un número fijo de ciclos por intervalo de tiempo establecido, de la misma manera que lo hacen los relojes usados por los seres humanos. Un reloj común, generalmente, produce un pulso, ciclo o tick cada segundo: tiene una *frecuencia* de un ciclo (o tick, si se prefiere) por segundo. Está claro que este reloj sólo puede medir intervalos mínimos de un segundo, si se desea medir intervalos menores a un segundo será necesario aumentar la frecuencia del reloj (más de un tick por segundo, por ejemplo).

La frecuencia se mide en *ciclos por segundo* (*c/sec o* $sec^{-1}$) o *hercio* o *hertz* (Hz). Así, por ejemplo, un reloj que mide segundos deberá tener, como mínimo, una frecuencia de 1 Hz o 1 c/sec y uno que mida décimas de segundo deberá tener una frecuencia de 10 Hz o 10 $sec^{-1}$.

Los relojes empleados en las computadoras deben producir muchos pulsos o ciclos por segundo ya que las computadoras ejecutan muchas instrucciones por segundo, del orden de varios miles. Una instrucción puede tardar, por ejemplo un micro segundo ($1" \times 10^{-6}$ o 0,000001", una millonésima de segundo) en ejecutarse. Esta velocidad se incrementa permanentemente ante la constante mejora de las tecnologías empleadas.

La frecuencia de los relojes de las computadoras es, entonces, muy alta: hay muchos pulsos por segundo. Los necesarios para medir el tiempo más corto que tarda la ejecución de la acción más corta que debe ejecutar la CPU (que no es necesariamente una instrucción completa sino parte de una). Esto hace que las frecuencias de éstos

relojes se midan en millones de pulsos por segundo: *megahercio* (MHz) o en *gigahercios* (GHz), miles de millones de pulsos por segundo; *terahercios*, $10^{12}$; *petahercios*, $10^{15}$ ; etc.

## PROCESADORES Y MICROPROCESADORES

A la CPU se la conoce, muchas veces, como el *procesador* de la computadora, quizás por ser la parte encargada de controlar el procesamiento de la información dentro de la computadora.

Por otro lado, es muy común hoy en día hablar de *microprocesadores*, y como extensión de *micro computadoras*, para referirse a aquellas máquinas que tienen microprocesadores.

En realidad, no existe ninguna diferencia substancial, o conceptual, entre los procesadores y los microprocesadores. Es sólo una diferencia de *empaquetado*. Todas las computadoras tienen CPU, o procesador. Hace unos años, por razones puramente tecnológicas, y porque, además, era posible, se pasó de fabricar los procesadores en una o más plaquetas con varios circuitos integrados, a un procesador hecho en un solo circuito integrado. A estos procesadores construidos en un solo circuito integrado se los llama microprocesadores.

## PROCESADORES Y COPROCESADORES

Las computadoras necesitan de un procesador (CPU) para funcionar. Este tiene todas las funciones, todas las instrucciones básicas, necesarias para realizar las operaciones más comunes. Aquellas que faltan pueden reemplazarse por programas hechos usando las instrucciones del conjunto de instrucciones que tiene la CPU.

Así por ejemplo, muchas CPU cuentan solo con la operación aritmética de la suma, pudiendo reemplazar con ésta, por medio de programas, las otras operaciones. Por ejemplo, la resta puede ser programada como una suma de números naturales, usando además la operación de módulo; la división como restas sucesivas y la multiplicación como sumas sucesivas.

Sin embargo algunas veces estas funciones no son suficientes. En efecto, por razones de velocidad (la mayor parte de las veces) es necesario incorporar otras instrucciones a la CPU. Generalmente, se trata de instrucciones aritméticas (resta, división, multiplicación y otras).

En lugar de incorporar las instrucciones a la CPU, lo que la hace más cara, una solución encontrada fue agregar otro procesador cuando resultase necesario. Este otro procesador, llamado *coprocesador*, trabaja bajo el control de la CPU y sólo tiene las instrucciones faltantes. Estos coprocesadores tienen diversos usos, los más comunes son aquellos que cuentan con instrucciones aritméticas, aunque existen otros que agregan instrucciones para el procesamiento de sonidos, imágenes, etc.

Sin embargo, muchos procesadores, que empezaron cómo coprocesadores, particularmente para procesamiento de imágenes y gráficos, y que necesitan de instrucciones aritméticas complejas, se emplean como procesadores de uso general.

## MEMORIA PRINCIPAL

La *memoria principal*, es donde la computadora almacena, o "memoriza" los programas y los objetos o datos sobre los que debe actuar para realizar las acciones para las que ha sido programada.

La memoria principal está compuesta de un número finito de *elementos de almacenamiento*. Cada uno de estos elementos de almacenamiento debe poder ser *accedido* por la CPU, ya sea para almacenar algo o para conocer lo que está almacenado. La acción de almacenar o guardar es conocida como *escritura* o *grabación*; la acción de recuperar, conocer, mirar lo que ya está almacenado, como *lectura*.

Para poder realizar estas acciones de escritura y lectura es necesario que cada uno de los elementos de

almacenamiento pueda ser identificado para que se distingan unos de otros. Esto se hace dándole a cada uno una dirección, que en este caso no es más que un número que le es propio y lo distingue.

Como la capacidad de "contar" de la CPU es finita, es decir que solo puede "contar" hasta un cierto número que depende de cada tipo de CPU, las direcciones de los elementos de memoria que la CPU puede direccionar dependerán de su capacidad de contar. Así una CPU que sólo puede "contar" hasta un millón, sólo podrá direccionar, acceder, un millón de elementos de memoria como máximo. Esto es conocido como el *espacio direccionable* de la CPU. Toda memoria que se coloque dentro de este espacio la llamaremos *memoria principal*. En realidad, cuando hablamos de la memoria principal de una computadora queremos referirnos a aquella memoria que la CPU puede *direccionar* en forma directa.

Dado que la memoria principal le sirve a la CPU para almacenar programas y datos, hay una estrecha relación entre la memoria principal de una computadora y la CPU de la misma. El programa, que se desea que la computadora ejecute, o por lo menos las partes que se estén ejecutando en un momento dado, debe estar guardado mientras se ejecuta en la memoria principal. Allí mismo, en la memoria principal, deben encontrarse los objetos o datos que sean necesarios para realizar las acciones del programa.

La interacción entre la memoria principal y la CPU es, entonces, permanente. Las acciones de ambas deben estar estrechamente sincronizadas: la memoria principal debe entregar a la CPU las instrucciones del programa una a una en el orden establecido en el programa. Asimismo, debe entregar los objetos o datos necesarios para ejecutar el programa.

Este sincronismo se obtiene porque ambos, CPU y memoria principal, "conocen" sus respectivas velocidades de funcionamiento, y en consecuencia se adaptan uno con el otro, empleando para ello el reloj que toda computadora tiene.

Como muchas cosas, tampoco la memoria principal es infinita en su *capacidad de almacenamiento*. Por el contrario tiene límites.

## UNIDADES DE MEDIDA DE ALMACENAMIENTO

La *capacidad de almacenamiento* de la memoria principal se mide en unidades de almacenamiento llamados *bytes*. Un *byte* es la unidad necesaria para almacenar un *carácter*.

Un *carácter* es un elemento de un conjunto de símbolos que se usan normalmente en la escritura como por ejemplo: una letra (de la *a* a la *z*, o de la *A* a la *Z*, ya que en este caso se hace una distinción entre los símbolos empleados para las minúsculas y las mayúsculas); los símbolos empleados para denotar los números del cero al nueve; símbolos de puntuación y otros, como por ejemplo: + ) @ : , ! ; . $ ( * % & - = ] # [ ? / ", entre otros.

Se emplea el prefijo *Kilo* para denotar las unidades de mil caracteres (*Kilobyte* o *KByte* o *KB*). Sin embargo, por razones de uso del sistema de numeración binario**,** en las computadoras, las unidades de mil bytes se equiparan con la potencia de dos más cercana ($2^{10}$ = 1.024) y así sucesivamente. Es decir que, por ejemplo, 2 Kb es igual a 2.048 caracteres y no a 2.000 caracteres.

Lo mismo sucede con las unidades de un millón de bytes, para el que se emplea el prefijo *Mega* (*Megabyte* o *MByte* o *MB*). Así, por ejemplo, 1 Mb es igual a 1.048.576 bytes ($2^{20}$ bytes); y 3 Mb es igual a 3.145.728 bytes.

Hace unos años atrás la capacidad de la memoria principal de las computadoras más comunes estaba en el orden de los Kb. Luego se medían en megabytes, hoy en día (2009) nos encontramos en el orden de los *gigabytes* (Gb) (1Gb = $2^{30}$ bytes =1.073.741.824 bytes). Es de esperar que en los próximos años tengamos que emplear los *terabytes* ($2^{40}$ bytes), (1Tb = 1.099.511.627.776 bytes), los *petabytes* ($2^{50}$ bytes); etc.

Para tener una idea de la capacidad de una memoria principal de 1 Mb, puede pensarse en un libro de 300 páginas, y para el que suponemos que cada página tiene 50 líneas, cada línea 10 palabras, y cada palabra 5 caracteres: $300 \times 50 \times 10 \times 5 = 750.000$ caracteres o bytes, aproximadamente 732 KB; es decir que no llega a 1 MB.

## TECNOLOGÍAS DE FABRICACIÓN

La tecnología empleada para fabricar los componentes electrónicos necesarios para las memorias de las computadoras es muy variada. Podemos reconocer dos grandes tipos: las memorias **RAM**, siglas en inglés de **R**andom **A**ccess **M**emory (Memoria de Acceso Aleatorio) cuya característica principal es que, cuando no está alimentada con energía, pierde (se borra) lo que tiene almacenado. Estas memorias pueden escribirse (grabarse) y leerse muchas veces.

Por otro lado están las memorias **ROM**, siglas en inglés de **R**ead **O**nly **M**emory (Memoria de Sólo Lectura), cuya característica principal es que no pierden lo que tienen almacenado cuando no están alimentadas por energía pero que, por el contrario, no pueden escribirse (o grabarse) más que una sola vez. Pueden, sin embargo, leerse muchas veces.

Existen otros tipos de tecnología para memorias, variantes de las ROMs, que pueden escribirse más de una vez aún cuando esta escritura es más complicada que en las RAMs, o que para borrarlas, antes de escribirlas, es necesario realizar algún proceso también más complicado que con las RAMs. Algunas de las más comunes son las PROM, EPROM, etc.

Un papel destacado en las *Computadoras Personales* (*Personal Computers*, PC, en inglés) lo ocupan las memorias construidas con tecnología MOS o CMOS, que sirven, aún cuando se apague la PC, para guardar sus características, la configuración de la misma como, por ejemplo, qué tipo y cuantos discos tiene, qué tipo de monitor, etc. Esto lo pueden hacer porque quedan alimentadas con una batería, que es suficiente, ya que justamente una de las principales características de estas memorias es su bajo consumo de energía.

## LAS PARTES EXTERNAS

Empleando una analogía con los seres humanos, las "partes externas" tienen que ver, por un lado, con los "sentidos" de la computadora y los órganos necesarios para contar con ellos para captar el mundo exterior: las *entradas*. Por otro lado, las partes externas tienen que ver, también, con los medios que utilice la computadora para accionar sobre el mundo exterior: hablaremos de las acciones (siguiendo con la analogía: hablar, escribir, andar, etc.) y diremos que son las *salidas* de la computadora, y que, siguiendo con la analogía, por medio de los órganos adecuados (cuerdas vocales, manos, piernas, etc.), acciona sobre el mundo exterior.

Claro está que los "órganos" necesarios para producir estas entradas y salidas no son iguales a los de los seres humanos, como que tampoco los "sentidos" son los mismos. La analogía vale solo para la comprensión inmediata del tema. Así hablaremos de *dispositivos de entrada/salida* de la computadora y no de órganos.

Es de hacer notar que, debido a la amplia difusión de las computadoras y su aplicación en distintas actividades, los dispositivos, tanto de entrada como de salida, son muchos más variados que los órganos de los seres humanos, por ejemplo, algunos sirven para detectar radiaciones (electromagnéticas y sonoras) que van más allá del rango que es accesible a los seres humanos y accionan con una precisión y velocidad varias veces mayor que estos. Puede decirse que la variedad de estos dispositivos se cuentan, quizás, en los miles. Dado el número tan grande de dispositivos, describiremos aquí solo algunos de los más comunes.

## TECLADO

El teclado ha sido y es, todavía, el medio de entrada, a la computadora, más común. Por lo menos para entrar

caracteres. El teclado es similar al de una máquina de escribir. ¿Cuánto tiempo ésta analogía será válida? ¿Cuántos de los jóvenes lectores han visto una máquina de escribir eléctrica o mecánica?

Un teclado consiste de un conjunto de teclas, donde cada una de ellas representa una letra del alfabeto, un dígito (del cero al nueve), además de los símbolos corrientes de puntuación y algunos otros.

Cada vez que se presiona una tecla, los circuitos del teclado envían a la computadora un código correspondiente al carácter que dicha tecla representa. Por supuesto, tanto el teclado como la computadora, ambos "conocen" la codificación correspondiente a cada tecla, de otra manera la comunicación sería imposible.

## VIDEO

Como el teclado ha sido y es el medio más común de entrada, la pantalla de video es y ha sido el medio más común de salida.

La *pantalla de video* comparte con la pantalla de los televisores su tecnología: ambos son *tubos de rayos catódicos* (*Cathode Ray Tube*, CRT, en inglés), aún cuando en ambos medios, televisión y computación, ya hayan aparecido otras tecnologías como por ejemplo el *cristal líquido* (*Liquid Crystal Display*, LCD, en inglés), o los LEDs (*Light Emiting Diode*).

Sin embargo, la pantalla, que se conoce generalmente con el nombre de *monitor*, no lo es todo. Al igual que un aparato de televisión que necesita del receptor y amplificador para poder mostrar por la pantalla las imágenes, una pantalla de computadora necesita de un dispositivo electrónico para poder mostrar las salidas que la computadora envía a la pantalla.

Este dispositivo es conocido comúnmente como el *controlador de video* o *plaqueta controladora de video* y es el encargado de recibir de la CPU las imágenes a proyectar en la pantalla y de enviar a ésta los impulsos electrónicos necesarios para producir dichas imágenes. El controlador tiene su propia memoria, llamada memoria de video, para almacenar temporalmente las imágenes que se envían al monitor.

Por supuesto, las imágenes pueden mostrarse tanto en *blanco y negro* (mal llamado muchas veces *monocromo*, un solo color) o en *color*. Para este último caso, se necesitan controladores y monitores condicionados para ello. De acuerdo con la cantidad de colores potenciales que pueden mostrarse se habla de la *palette* de colores disponibles, esto no es lo mismo que la cantidad de colores que pueden mostrarse al mismo tiempo en la pantalla.

En efecto, se pueden tener disponibles varios millones de colores, por ejemplo dieciséis millones, pero solo poder mostrarse, por la pantalla, unos pocos al mismo tiempo, dieciséis o doscientos cincuenta y seis, dependiendo del controlador y del monitor.

La *calidad de imagen* del conjunto de controlador de video y monitor se mide en la *cantidad de puntos* que pueden mostrar. Cada uno de estos puntos se lo conoce con el término *pixel* en inglés. En efecto, la pantalla puede verse como un rectángulo lleno de pequeños puntos que pueden estar encendidos o no de acuerdo con la imagen que se desee formar. Cuantos más puntos se puedan mostrar en una superficie dada mayor será la calidad de la imagen. Es decir que cuanto mayor sea la densidad de puntos, para una superficie dada, mayor será la resolución. Hay que tener en cuenta que muchas veces se emplea cantidad de pixels para referirse a la resolución. La cantidad de pixels se mide como el producto del ancho por el alto, ambos medidos en cantidad de puntos, que se pueden mostrar. Sin embargo, la resolución es la cantidad de puntos por la superficie de la pantalla. Es decir la misma cantidad de puntos dará una mayor resolución cuánto más pequeña sea la superficie. Esto es la densidad. Es de hacer notar que la resolución incluye otras características además de la densidad.

Cada píxel tiene una serie de atributos, como por ejemplo el *tono* en el caso del color y en algunos casos de blanco y negro que se transforman en tonos de grises, el *parpadeo*, la *inversión* (de blanco a negro o viceversa), el *subrayado*, etc.

Los caracteres o símbolos (letras, dígitos, signos de puntuación, etc.) para ser mostrados en la pantalla, al igual que cualquier otra imagen, se construyen con estos puntos o pixeles.

Muchas veces, en especial en las PCs, se distinguen dos modos de mostrar las salidas en la pantalla: *modo texto*, que es cuando no se pueden mostrar figuras sino solo símbolos de escritura, de texto, como letras, dígitos (del cero al nueve), signos de puntuación y *modo gráfico*, que es cuando se puede mostrar cualquier tipo de imagen.

A modo de ejemplo damos una tabla con algunos de los estándares más comunes de las PCs. Algunos de estos estándares ya ha caído en desuso, por ejemplo: Hercules (no lleva acento porque está escrito en inglés), CGA, EGA.

Tabla 1.1. Características de los sistemas de video.

| Estándar | Observaciones | Modo Gráfico ancho × alto (pixels) |
|---|---|---|
| HERCULES | | b/n 720 × 348 |
| CGA | En Modo Texto: carácter: (pixels) ancho × alto: 8 × 8 CRT: líneas × columnas: 40 x 25 ó 80 x 25 ó 132 x 25/44 | b/n 640 × 200; 4 c 320 × 200 |
| EGA | carácter: (pixels) ancho × alto: 8 x 14 | 64 c 640 x 350 |
| VGA | | 640 x 480 video/palette: 32.768/32.768 ; 256/256.000 ; etc. |
| SUPER VGA O XGA | | 800 × 600; 256/256.000; etc. 1.024 × 768; 256/256.000 ; etc. |
| SXGA+ | | 1400 × 1.050 |
| UXGA | | 1.600 × 1.200 |
| WXGA | Usado en general para wide-screens, relación similar a pantalla de cine: 16:10 o 16:9 en lugar de 4:3 | 1.440 × 900 |
| WUXGA | | 1.920 × 1.200 |

## MOUSE (RATÓN)

Otro dispositivo de entrada a la computadora ampliamente difundido es el *mouse* (en Inglés) o *ratón*. Este dispositivo comanda el cursor que se posiciona en la pantalla de video en los ambientes gráficos. Para comandar el movimiento del cursor se utilizan distintos mecanismos en el mouse. Algunos son electromecánicos: el usuario al mover el mouse sobre una superficie hace que una bola ruede entre dos rodillos colocados a 90º, a su vez el movimiento de estos rodillos es detectado electrónicamente lo que permite determinar la dirección y distancia del movimiento del mouse. Existen también algunos ratones que emplean un haz de luz cuyo reflejo sobre una superficie es captado electrónicamente y es esto lo que permite en este caso determinar la dirección y distancia del movimiento del mismo.

## MEMORIA AUXILIAR

Como en muchos casos de la vida corriente, nadie está satisfecho con lo que tiene y las computadorass no son una excepción: no les basta con la memoria principal y necesitan de *memoria auxiliar*.

En realidad, la diferencia más importante entre la memoria principal y la memoria auxiliar es que la primera está íntimamente ligada a la CPU, ya que como hemos visto, la CPU puede *direccionarla directamente* (está dentro de su *espacio direccionable*), mientras que la segunda no. Se podría decir que la memoria principal es comparable a la memoria de los seres humanos: “reside” en el cerebro de éstos.

La memoria auxiliar podría comparársela con un cuaderno, una agenda, etc., que puede emplearse a voluntad y

que no es estrictamente necesario para el funcionamiento. Además es extensible: siguiendo con la analogía, pueden comprarse tantos cuadernos, agendas, etc., se necesiten a medida que sean necesarios. Lo mismo pasa con la memoria auxiliar de las computadoras.

La *capacidad* de la memoria auxiliar se mide, igual que la de la memoria principal, empleando los bytes o caracteres.

Hay distintos tipos de memoria auxiliar; las más comunes son las que se encuentran sobre un *soporte magnético* (a diferencia de los cuadernos, por ejemplo, que están sobre un soporte hecho de papel). De éstas, hay dos clases bien diferenciadas: las *cintas* y los *discos*. La diferencia fundamental entre una y otra es la forma de *acceso* a lo que se encuentra almacenado. En las cintas el acceso es *secuencial*, como en los antiguos casetes de música o sonido donde había que recorrer toda la parte de la cinta que precedía lo que se deseaba escuchar. En los discos el acceso es *directo*; al igual que los discos de música, sean estos de pasta –long plays– u ópticos –láser o compact disks, donde puede accederse directamente a lo que se quiere escuchar, acá también puede accederse de forma directa a lo que se encuentra almacenado.

En los últimos años han aparecido también para las computadoras los discos ópticos, conocidos como DVDs (*Digital Versatile Disc* o *Digital Video Disc*) y CD-ROM (*Compact Disk ROM*), porque como se escriben (graban) una sola vez y se leen muchas, se parecen en esto a las memorias ROM. Aún cuando ya haya CDs y DVDs que pueden leerse y grabarse varias veces.

Hay también discos optomagnéticos, es decir que emplean ambas tecnologías, óptica y magnética, generalmente la óptica sirve para posicionar la cabeza lectograbadora sobre la superficie del disco y la magnética para grabar y leer.

## DISCOS MAGNÉTICOS

Los discos, especialmente los magnéticos, se han constituido en el medio más comúnmente empleado en las computadoras. Estos están hechos sobre una base que tiene una capa de un material que puede soportar un campo magnético sobre el que se graba la información. Esta información se graba, entonces, empleando propiedades de los campos magnéticos.

El funcionamiento de los discos es básicamente muy simple: cuentan con un *dispositivo de lectura y grabación* que hace que los mismos giren a una cantidad de revoluciones por minuto establecida, mientras que una cabeza de lectura y grabación se desplaza, con movimientos radiales o de otro tipo, sobre la superficie para efectuar la operación deseada de grabación de información o lectura de información ya almacenada.

El disco está conectado a la CPU por medio de un conjunto de circuitos electrónicos especializados, que se encargan de toda la operación de la unidad de lectograbación y que se conocen con el nombre genérico de *Controlador de Disco*. Hay distintos tipos de controladores para los distintos tipos de dispositivos de lectograbación, capacidad de almacenamiento y velocidad de transmisión de éstos.

En efecto, la *capacidad total* de un disco y/o su unidad de lectograbación puede medirse en términos de su *capacidad de almacenamiento* y la *velocidad de lectura y grabación*, es decir la velocidad con que envía y recibe información de la CPU.

Existen distintos tipos de discos: *flexibles*, también llamados *disquetes* o *floppy disks* en inglés, cuya base es flexible, blanda. Otros son *rígidos* o *duros*, porque su base es dura, rígida. No debe confundirse la funda que muchas veces envuelve a los discos, que en algunos casos puede ser flexible y en otros rígida, aun cuando dentro haya un disco flexible.

Existen, por otro lado, discos *fijos* que se encuentran montados dentro del dispositivo de lectograbación en la computadora y que no pueden quitarse a menos de una intervención de personal especializado y en general por

razones de reparación. Por otro lado, hay discos *removibles* por el usuario común de la computadora y que en el curso normal de operación de la computadora podrán ser puestos o quitados del dispositivo de lectograbación. Es importante, entonces, distinguir entre el *dispositivo* de lectograbación (el equivalente de un "grabador" o "reproductor" de casetes de audio) y el *medio* sobre el que se almacena la información (el equivalente del casete).

Los discos magnéticos están organizados en *caras* o *lados* (*faces*, en inglés), éstos a su vez en *pistas* (*tracks* en inglés) concéntricas (no en un espiral como los discos de pasta de música) las cuales están divididas en *sectores*, que es donde se graban los bytes o caracteres. Generalmente, un disco tiene dos caras o faces. Sin embargo, hay dispositivos de lectograbación que aceptan discos apilados (armados de esa manera) por lo que hay más de dos caras en cada unidad que se pone en el dispositivo de lectograbación. Las pistas que se encuentran en cada cara en la misma posición constituyen lo que generalmente se conoce como un *cilindro*.

Los discos que vienen directamente del fabricante deben ser procesados antes de poder ser empleados en la computadora. Este proceso, que se conoce con el nombre de *formateado* es de características distintas dependiendo del tipo de disco, de la computadora y sus programas básicos (Sistema Operativo).

### DISCOS MAGNÉTICOS FLEXIBLES (FLOPPY Ó DISQUETE)

En general, los disquetes son *removibles*, es decir que pueden ponerse y sacarse del dispositivo de lectograbación en el curso normal de operación de la computadora. Al igual que los casetes, éstos tienen un sistema de protección de grabación.

Hay distintas *medidas* para los disquetes. Se los puede medir de acuerdo a su *diámetro*, expresado en pulgadas (por ejemplo, 8, 5¼, 3½, 3, 2), o a su *capacidad de almacenamiento*. Los más comunes son de 5¼ de 360 Kb y de 1,2 Mb. De 3½ de 720 Kb y de 1,44 Mb.

Los disquetes emplean una o dos caras dependiendo de la capacidad de los mismos. Tienen entre 40 a 80 pistas y 8, 9 o 15 sectores por pista, con una capacidad de 128 a 1.024 caracteres por sector, siendo el de 512 el más común.

En la actualidad estos dispositivos han caído en desuso y se han reemplazado por las memorias de estado sólido, con tecnología similar, aunque no igual, a la tecnología empleada para la fabricación de la memoria principal, que se encuentran en los dispositivos de memoria USB (pen drives, etc.)

### DISCOS MAGNÉTICOS RÍGIDOS (DUROS)

En general los discos rígidos o duros no son removibles, aún cuando hay dispositivos que sí lo permiten pero no son los más comunes. Es decir que el disco se encuentra sellado dentro de la unidad de lectograbación.

La capacidad de almacenamiento de estos discos es mucho mayor que la de los disquetes: va de los Megabytes a los Gigabytes. Su velocidad es también varias veces mayor que la de los disquetes.

## IMPRESORA

La impresora constituye otro de los periféricos más comunes de las computadoras. Ella sirve para obtener información impresa sobre papel. Es, como la pantalla, un dispositivo de salida, siendo ésta no sobre una pantalla de rayos catódicos, sino sobre papel.

Existen diversas tecnologías de impresoras. Una clasificación posible es:

Impresoras de *impacto*, que son las que impactan, golpean, mecánicamente sobre una cinta entintada y sobre el papel para imprimir los caracteres, y que a su vez pueden separarse en: *impresoras de línea*, que son las que imprimen una línea por vez, todos los caracteres al mismo tiempo, y de las que existen varias tecnologías (tambor, cinta, etc.); o *a carácter*, que imprimen un carácter a la vez; estas también emplean distintas técnicas (bocha:

similar a ciertas máquinas de escribir mecánicas o electromecánicas, cilindro, roseta o margarita, agujas: las más comunes).

Impresoras de *deposición de tinta*, que colocan, depositan, la tinta sobre el papel, sin emplear un golpe para ello. Las tecnologías empleadas para ello son variadas: *chorro de tinta*: que arroja minúsculas gotas de tinta para dibujar el carácter a ser impreso; *láser*: que emplea un láser u otra fuente de "luz coherente" para depositar la tinta electroestáticamente sobre el papel (similar a las fotocopias). En éste grupo podemos incluir a las impresoras que emplean un papel termo sensible (que reacciona al calor) y que por medio de agujas calientes puede dibujar los caracteres sobre el papel (método empleado por los faxes), aunque, debido al costo y a la calidad de la presentación del escrito final, han caído en desuso en las computadoras.

La *capacidad* de una impresora puede considerarse desde varios aspectos: una impresora de impacto, que tenga una matriz con el formato de cada carácter, imprimirá caracteres muy nítidos (¡sobre todo cuando está nueva!) pero no podrá cambiar ni la fuente (por ejemplo `Courier`) ni el tamaño de los mismos. Tampoco podrá imprimir otra cosa que no sean caracteres: no podrá, por ejemplo, imprimir figuras ni gráficos. Por el contrario las impresoras que tienen una aguja o un chorro de tinta y que construyen, dibujan, cada carácter al imprimirlo, podrán, de la misma manera, imprimir figuras y gráficos.

Otro aspecto a tener en cuenta es la necesidad de color, ya que hay impresoras que no solo imprimen con un solo color de tinta sino que pueden hacerlo con varios colores distintos.

Debe pensarse en el tipo de papel que emplea la impresora (por ejemplo si es termo sensible o papel común, etc.) así como el tamaño máximo de papel que puede emplear.

Es importante considerar la *velocidad* de impresión, que se mide de distintas formas: en las impresoras de línea en *líneas por minuto*, de seiscientas *lpm*, por ejemplo. En las de carácter en *caracteres por segundo*, de trescientos *cps*, aunque esto varía mucho de acuerdo con la calidad de impresión. La velocidad de las impresoras láser o de chorro de tinta suele medirse en *páginas por minuto*.

La *precisión* (algunas veces también llamada *calidad de impresión*) de una impresora es importante ya que hace a la claridad y a la elegancia de la escritura así como a la precisión con que figuras y gráficos son impresos. Esto tiene sentido en aquellas impresoras, que como las de chorro de tinta o láser, construyen sus impresos sobre la base de puntos (tanto los caracteres como las figuras). Por eso mismo, la precisión suele medirse en *puntos por pulgada*, es decir cuántos puntos puede poner en una pulgada. Las impresoras láser y de chorro de tinta más comunes hacen aproximadamente trescientos o más puntos por pulgada.

## 1.3 LA INFORMACIÓN DENTRO DE LA COMPUTADORA

Ya hemos dicho que la tarea principal de una computadora es la de ejecutar programas. Pero ¿qué *hacen* estos programas cuando se ejecutan dentro de la computadora? La respuesta es que *procesan* información. Información que ya tienen porque se la ingresaron previamente: o bien porque es el producto de un proceso anterior realizado por la misma computadora, o porque ha sido ingresada durante la ejecución del programa.

Esta información es en realidad una *representación* de cosas del mundo externo a la computadora. Así cuando una computadora procesa la información sobre el personal de una empresa para liquidar sueldos, por ejemplo, o procesa la información sobre consumo de electricidad de los abonados de una empresa eléctrica para producir las facturas, tiene la información necesaria para ello (¡con suerte sin errores para que no se produzcan facturas exorbitantes o sueldos ridículos!). Obviamente, ni los empleados de la empresa a los que hay que liquidar sueldos, ni el dinero para pagarles, ni los kilowatt del consumo eléctrico de los abonados están realmente dentro de la computadora. Solamente se encuentra la información que *representa* dichas cosas. Esta información, esta representación abstracta de aspectos de la realidad, se conoce con el nombre de *datos*.

Son los datos, por ejemplo, sobre los empleados (nombre, dirección, categoría, etc.) necesarios para el proceso

de liquidar sueldos los que se encuentran almacenados en la computadora cuando se realiza su liquidación de haberes.

Estos datos se encuentran en la memoria auxiliar de la computadora (sobre todo cuando son muchos) y pasan a la memoria principal a medida que son requeridos por la CPU para su procesamiento.

Por supuesto que dichos datos deben estar organizados dentro de la computadora para que puedan ser utilizados. Esta organización depende, claro, de la finalidad que se desee. Así los datos sobre los empleados de una empresa estarán agrupados de manera tal que puedan ser usados para liquidar sueldos si es eso lo que se desea. La decisión sobre el tipo de organización depende del usuario de la computadora y del fin que pretenda. El sistema computacional le permitirá, por medio de programas específicos realizar dicha organización.

Generalmente los datos se agruparán, en una primera instancia, en *archivos* (files, en inglés). Siguiendo con el ejemplo de los empleados, podría existir un archivo que almacene los datos de todos y cada uno de los empleados. Este archivo tendrá una identificación de manera tal que los programas que deban procesar los datos puedan reconocerlo. La mayor parte de las veces la identificación consistirá de un *nombre*, pero puede haber otros tipos de identificación; así, archivos con nombres iguales podrían distinguirse por medio de la fecha y hora en la que fueron guardados en la computadora, etc.

Los archivos podrán, a su vez, agruparse en otros grupos, que se conocen como *directorios*, o cuentas, de manera tal que se pueda organizar mejor la información. Esto sería equivalente a tener organizada la información sobre los empleados de una empresa, por ejemplo, en archivos que agrupen a los empleados de las distintas secciones de una sucursal, y a su vez tener un grupo con todas las secciones. Por supuesto estos agrupamientos pueden realizarse en función de lo que se desee agrupar y pueden continuarse tanto como sea necesario y se tenga capacidad para hacerlo.

Existen distintos *tipos de datos;* en efecto, las computadoras pueden procesar distintos tipos de información, no sólo la que representa a los empleados de una empresa tales como el nombre (que son letras), su edad (números), sino también puede procesar la foto del empleado (que es una imagen). Es decir, los archivos pueden almacenar datos de distintos tipos.

Pese a que podemos almacenar y procesar en una computadora distintos tipos de datos, las computadoras sólo almacenan y procesan números. ¿Cómo es esto posible, entonces? Porque las computadoras *codifican* la información que se les da. Transforman, por ejemplo, las letras, las imágenes, los sonidos, en números.

Es fácil imaginar cómo es posible esto: por ejemplo podríamos codificar las letras del abecedario diciendo que a la letra *A* le corresponde el número uno, a la *B* el dos y así sucesivamente. Habría que codificar también la letra *a* y todas las minúsculas de la misma manera que se codificaron las mayúsculas. Pero eso no es problema ya que tenemos infinitos números. De esta manera se puede tener, entonces, dentro de una computadora la representación (la codificación) de letras, números, símbolos, imágenes, sonidos, etc. Conociendo la codificación correspondiente se puede restituir la información original. De esa manera, por ejemplo, se pueden almacenar números, que surgen de la transformación y codificación de sonidos musicales correspondientes a una canción dada, y luego reproducir dichos sonidos, realizando la transformación inversa: se descifran (o como algunas veces se dice se decodifican) los números y se los transforma en sonidos.

En realidad, si bien las computadoras se han diseñado para que empleen números, la notación de los mismos o el sistema de representación empleado, no es el decimal, que es el comúnmente usado por los seres humanos, sino que por razones tecnológicas el sistema numérico empleado es el *binario.* El sistema binario tiene como base el número dos, ya que es fácil hacer corresponder, electrónicamente, los dos únicos números que puede emplear (el cero y el uno) con la existencia o no de corriente, o con dos voltajes distintos, etc.

Existen distintas formas de codificar los caracteres (letras, dígitos decimales -0 al 9-, símbolos de puntuación, etc.), dentro de las computadoras. Una de las más utilizadas en los últimos tiempos es la llamada ASCII, que se pronuncia “asqui” y cuyas siglas corresponden a las iniciales de *American Standard Comunication Information*

*Interchange*. En el Apéndice B se muestra una tabla conocida como la *tabla ASCII* que muestra la correspondiente codificación para 128 caracteres. Emplea los números del cero al ciento veintisiete (ciento veintiocho códigos), para lo que se necesitan siete dígitos en la numeración binaria. *La tabla ASCII extendida* emplea doscientos cincuenta y seis códigos, del cero al doscientos cincuenta y cinco, necesitando así ocho dígitos en la numeración binaria para representar el número más grande de esta codificación, el 11111111.

Como las computadoras sólo trabajan con números, las tablas de caracteres, como la ASCII, contienen los códigos numéricos asignados no sólo a los caracteres imprimibles como, por ejemplo 'a' o '+' sino también aquellos asociados a caracteres de control como por ejemplo el 10 que corresponde a avanzar una línea en una impresora. Estos últimos –que son los primeros treinta y dos– se conocen como caracteres "no imprimibles" o de control, es decir que no se los ve. Hoy en día no son muy utilizados, además la descripción de los mismos (lo que hacen) no es muy clara ya que la tabla ASCII apareció hace ya mucho tiempo (alrededor de 1963) y se usaba para los teletipos por lo que muchos de esos caracteres "no imprimibles" eran para acciones propias de los teletipos. Esta tabla es también la adoptada por la *International Standards Organization* (ISO) y se la conoce como ISO 8859 o ISO 10646, que es la nueva norma de ISO para compatibilizarla con UTF-8 que es la norma del consorcio Unicode.

Tradicionalmente, entonces, se emplean los ocho dígitos binarios necesarios para representar los caracteres de la tabla ASCII, como un *byte*. Un byte tendrá entonces la capacidad de codificar, o representar, doscientos cincuenta y seis números (del cero al doscientos cincuenta y cinco, generalmente), por lo que necesitará de ocho dígitos binarios. Cada uno de estos dígitos se conoce con el nombre de *bit*, de ***b****inary dig****it*** (dígito binario, en inglés, algunos emplean *dio*, de ***d****ígito binar****io***), por lo que generalmente se dice que un byte está compuesto de ocho bits.

Otras codificaciones, introducidas para representar otros alfabetos (griego, árabe, cirílico, etc.) tales como Unicode y Conjunto de Caracteres Universal (UCS) ISO/IEC 10646 definen un conjunto de caracteres mucho mayor, y sus diferentes formas de codificación han empezado a reemplazar ISO 8859 y ASCII rápidamente en muchos entornos.

## 1.4 EL FUNCIONAMIENTO

Cada vez que se enciende una computadora, ésta solo se dedicará a *ejecutar* un programa. Esta es la única tarea que las computadoras saben realizar. Pero sin embargo pueden, o parecen poder, hacer muchas otras cosas. En realidad esta capacidad de realizar las más diversas tareas es producto de los programas que las computadoras ejecutan.

Son los programas los que le *dicen* a la CPU, y en consecuencia a toda la computadora, qué debe hacer. Sin estos no habría actividad.

Utilizaremos indistintamente los términos *programa* y *sistema*. Generalmente la distinción hecha entre ambos es una de cantidad y no de calidad. En efecto, se reserva la palabra sistema para aquellos programas, que además de constituir una unidad de propósito, son grandes, reservando la palabra programa para los que son más pequeños. Como las cualidades de grande y pequeño son altamente subjetivas es que emplearemos uno u otro indistintamente.

La ejecución de un programa, instrucción tras instrucción, es tarea propia de la CPU. Estas series de instrucciones le dirán a ella que hacer, no solo con los datos que tenga, sino también con los distintas partes y dispositivos que se encuentran formando parte de la computadora.

Las instrucciones de los programas, así como los datos sobre los que estas deban actuar, deben encontrarse en la memoria principal de la computadora. Sin embargo, ésta puede no tener la capacidad para almacenar, de una sola vez, todas las instrucciones y datos necesarios para la ejecución de un programa. Esto se resuelve almacenando el programa y sus datos en la memoria auxiliar (discos, cintas, etc.) y *trayendo* lo necesario (instrucciones y datos) a la memoria principal a medida que se los necesite.

El *arranque* (*boot* o *bootstraping*, en inglés) de las computadoras merece un párrafo aparte. En efecto, apenas

encendida, la CPU ejecutará instrucciones, ¿pero qué instrucciones? Las necesarias para que de ahí en más la computadora esté lista para seguir funcionando. Estas son escogidas juiciosamente por el fabricante de la computadora para que esto suceda y el programa correspondiente es colocado en una parte de la memoria principal para que cada vez que se encienda la computadora comience a funcionar correctamente. Este programa se encargará de traer desde la memoria auxiliar el resto de los programas o sistemas necesarios para continuar con el funcionamiento de la computadora, usualmente el sistema operativo.

En muchas computadoras *casi* el mismo efecto puede lograrse si se tiene un botón o interruptor para efectuar lo que se llama, en inglés, *reset*, que podemos decir que significa *poner a cero* las distintas partes de la computadora. Decimos que es "casi lo mismo", aún cuando internamente no es exactamente lo mismo, porque por ejemplo no se borra la memoria principal de la computadora, cosa que sí sucede cuando se la apaga.

Podemos decir que, entre los programas o sistemas que se encuentran más frecuentemente en las computadoras, el más común y necesario es el que se conoce como *sistema operativo*, que es el encargado de administrar y hacer funcionar adecuadamente los distintos dispositivos y partes de la computadora y además de permitir que otros programas o sistemas puedan servirse de ellas. Sin un programa o sistema que se encargue de estas tareas sería imposible que la computadora pudiese funcionar en forma útil.

Esto es así aún cuando el sistema operativo no sea de uso general, sino orientado a una función específica, porque así lo requiere el destino de la computadora como por ejemplo, un sistema operativo encargado de administrar una computadora encargada del funcionamiento de una máquina de lavar platos. En estos casos, en general, no se habla de sistema operativo sino de un sistema orientado, reservando el término sistema operativo para aquellos sistemas de software que sirven para administrar una computadora de uso más general.

Es el sistema operativo el encargado de relacionar y coordinar el funcionamiento y la administración de la máquina y de sus partes: CPU, memoria, teclado, pantalla, impresora, memoria auxiliar, discos, etc., así como de la administración de los archivos, directorios, cuentas, etc., que se encuentran en la memoria auxiliar de la computadora.

Existen distintos tipos de sistemas operativos. Están los *monousuarios*, que administran las computadoras suponiendo que las mismas serán usadas por un solo usuario por vez y los *multiusuarios*, que administran las computadoras suponiendo que las mismas serán usadas por más de un usuario por vez. Se los puede clasificar también de acuerdo a la cantidad de tareas que pueden realizar a la vez: Los *monotarea*, que pueden ser monousuario o multiusuario, y que administran las computadoras permitiendo al o a los usuarios de las mismas, realizar sólo una actividad o *tarea* (ejecutar un programa o sistema) por vez, y los *multitareas*, que pueden ser monousuario o multiusuario, y que administran las computadoras permitiendo al o a los usuarios de las mismas, realizar más de una actividad o *tarea* (ejecutar un programa o sistema) por vez.

Por supuesto, los sistemas operativos multiusuarios deben permitir que más de un usuario trabaje sobre la computadora al mismo tiempo. Esto se hace conectando a la máquina *terminales*, que son dispositivos de entrada y salida formados, generalmente, de un teclado y una pantalla. Habrá, entonces, tantos usuarios usando la computadora al mismo tiempo como terminales tenga ésta conectadas.

Distinto es el caso de las llamadas *redes* de computadoras donde varias computadoras están interconectadas entre sí y permiten que los usuarios respectivos puedan comunicarse, intercambiando datos. Aquí hay varias computadoras conectadas, y no una sola con varias terminales.

El sistema operativo está compuesto de distintas partes, subsistemas o programas que se encargan de las distintas tareas que el sistema operativo debe llevar adelante. Las partes que administran los dispositivos de entrada y salida de datos, de memoria auxiliar, son generalmente conocidas con el nombre genérico, en inglés, de *drivers* (conductores o chóferes) ya que se suponen que *conducen* a dichos dispositivos.

Los sistemas operativos son, en general, propios a una familia de computadoras y varían en sus prestaciones (aún

cuando todos tengan un conjunto de características básicas esenciales) de una computadora a otra, y para las mismas computadoras, de un fabricante a otro. Están en permanente cambio y actualización, no sólo para seguir los cambios de las computadoras, sino para corregir defectos. Estas actualizaciones se conocen con el nombre de *versiones*, que están generalmente numeradas para indicar su aparición en el tiempo.

Los sistemas operativos son conocidos como *software de base*, justamente por *estar* en la *base* del funcionamiento de las computadoras.

Existen otros sistemas o programas que tienen funciones básicas, aún cuando no tan estrictamente necesarias como los sistemas operativos. Ellos son por ejemplo los programas de comunicación y redes, que permiten a las computadoras comunicarse entre sí; de diagnóstico de error; etc. Algunas veces dichos sistemas integran de una manera u otra el sistema operativo por lo que también son conocidos como software de base.

Esto pasa también con los *compiladores*, que son sistemas encargados de *traducir* los programas escritos en un lenguaje de programación, al lenguaje que entiende la CPU, para que así, el programa pueda ser ejecutado.

Con el nombre global de *utilitarios* se conocen una serie de sistemas que permiten realizar distintas tareas que ya se han transformado casi en tradicionales en el ámbito de la computación: *editores de texto*, que como su nombre lo indica permiten a los usuarios de las máquinas, editar (escribir, corregir, cambiar, etc.), con mayor o menor facilidad, un texto. Planillas de cálculo, administradores de bases de datos, y otros también pueden encuadrarse en esta categoría.

## 1.5 EL SISTEMA OPERATIVO

El sistema operativo es el programa que permite administrar las distintas partes de la computadora y hace que un usuario pueda realizar esto por medio de *comandos*, que son órdenes que los usuarios puedan darle para realizar las distintas operaciones.

Los comandos son, generalmente, en la mayor parte de los sistemas operativos, de dos tipos: *internos* y *externos*. Los internos son aquellos que están incorporados al *núcleo* (algunas veces se lo llama *kernel*, en inglés) del sistema operativo que es la parte central del mismo y podría decirse la que contiene lo mínimo necesario para que la computadora pueda operar. Los externos son aquellos que se pueden ir incorporando, como archivos de programa separados a medida que son necesarios. El sistema operativo, por medio de su núcleo, sin embargo ejecuta todos ellos.

Los comandos tienen una *sintaxis*, una ortografía propia que hay que respetar, ya que le sirve al sistema operativo para reconocerlos. Algunos de estos comandos pueden tener *parámetros*, es decir objetos sobre los que el comando operará. Por ejemplo, si se desea borrar un archivo, *sacarlo* del sistema, habrá que dar el comando correspondiente y el parámetro será el nombre del archivo que se desea borrar. Algunos comandos tienen un valor del parámetro por *defecto* (*default*, en inglés), es decir que si no se da ningún valor el sistema operativo *asume* uno. Dentro de los parámetros pueden emplearse caracteres *comodines* (*wild cards*, en inglés), que como la carta del juego de cartas así llamada puede tomar cualquier valor.

Algunos sistemas operativos permiten que un grupo, o un *lote* (*batch*, en inglés), de comandos puedan ser incorporados a un archivo y ejecutados todos, uno detrás del otro, ejecutando el archivo que los contiene como si fuese un programa. Es decir que se pueden armar programas de comandos para que sean ejecutados sin necesidad de *entrarlos* uno a uno cada vez que se los necesite.

Hay sistemas operativos que tienen incorporados mayores facilidades para esto ya que tienen un lenguaje de programación que va más allá de los comandos, permitiendo realizar gran cantidad de tareas en forma mucho más ágil y dinámica.

A lo largo del tiempo, la *interfaz del usuario*, que es como generalmente se conoce el conjunto de comandos que permiten que los usuarios y el sistema operativo se relacionen, ha cambiado. De los sistemas operativos basados en

comandos escritos se ha pasado a comandos puramente gráficos, simbólicos, que representan sus comandos por medio de *iconos*, pequeñas figuras que representan simbólicamente la acción deseada. Estas interfaces son conocidas como GUI, de las siglas en inglés **G**raphical **U**ser's **I**nterface (*Interfaz Gráfica del Usuario*). En algunos casos, cuando los sistemas operativos no han sido diseñados originalmente con este tipo de interfaz, se agrega dicha interfaz al sistema operativo como una *cáscara* (*shell*, en inglés) *alrededor* o *por encima* del sistema operativo.

Los sistemas operativos para que funcionen sobre una computadora determinada deben *conocer* como esta está compuesta, deben conocer su *configuración*, las partes que la componen y sus características. De esta manera el sistema operativo podrá incorporar los drivers correspondientes a cada dispositivo que necesite administrar. Muchas veces todas estas características estarán agrupadas en un archivo que se conoce con el nombre genérico de archivo de configuración o de instalación.

Cada dispositivo podrá tener, además de su nombre físico (por ejemplo: disco, marca, modelo, etc.), un nombre lógico (por ejemplo: *pepe*), que le servirá al sistema operativo para reconocerlo cuando los usuarios lo nombren. Todas las tareas que sirven para definirle la computadora y sus partes al sistema operativo y cómo se quiere que funcionen, se conocen generalmente con el nombre de *instalación*, ya que la operación es la instalación del sistema operativo en la computadora.

Cada vez que la computadora arranque empezará a funcionar ejecutando el sistema operativo. Este a su vez deberá realizar una serie de acciones llamadas de *inicialización*, que le permitirán seguir funcionando. Muchos sistemas operativos permiten que los usuarios puedan definir algunas acciones iniciales para que puedan ejecutarse en el momento del arranque. Estos son archivos especiales, de comandos, que el sistema operativo reconoce que deben ejecutarse inicialmente, en el arranque de la máquina.

# 2 PROGRAMAR

## 2.1 INTRODUCCIÓN

¿Qué es programar? La Real Academia de la Lengua Española (http://www.rae.es/) da la siguiente definición:

> *programar.* tr. Formar programas, previa declaración de lo que se piensa hacer y anuncio de las partes de que se ha de componer un acto o espectáculo o una serie de ellos. || 2. Idear y ordenar las acciones necesarias para realizar un proyecto. U. t. c. prnl. || 3. Preparar ciertas máquinas por anticipado para que empiecen a funcionar en el momento previsto. || 4. Preparar los datos previos indispensables para obtener la solución de un problema mediante una calculadora electrónica. || 5. *Inform.* Elaborar programas para la resolución de problemas mediante ordenadores. || 6. *Mat.* Optimar el valor de una función de muchas variables cuyos valores extremos son conocidos."

Vemos que la acepción 5 hace referencia a la Informática y es la que se aplica específicamente en nuestro caso. Sin embargo, la primera acepción es interesante también: "Formar programas, previa declaración de lo que se piensa hacer y anuncio de las partes de que se ha de componer un acto o espectáculo o una serie de ellos." Aún cuando no haya una referencia directa a la Informática, la actividad de programación está directamente relacionada con muchos aspectos de la vida corriente, no sólo con la Informática.

Si programar es "Elaborar programas ...", ¿qué es un *programa*? Nuevamente la Real Academia de la Lengua Española nos puede ayudar:

> *programa*. (Del lat. *programma*, y este del gr. πρόγραμμα). m. Edicto, bando o aviso público. || 2. Previa declaración de lo que se piensa hacer en alguna materia u ocasión. || 3. Tema que se da para un discurso, diseño, cuadro, etc. || 4. Sistema y distribución de las materias de un curso o asignatura, que forman y publican los profesores encargados de explicarlas. || 5. Anuncio o exposición de las partes de que se han de componer ciertos actos o espectáculos o de las condiciones a que han de sujetarse, reparto, etc. || 6. Impreso que contiene este anuncio. || 7. Proyecto ordenado de actividades. || 8. Serie ordenada de operaciones necesarias para llevar a cabo un proyecto. || 9. Serie de las distintas unidades temáticas que constituyen una emisión de radio o de televisión. || 10. Cada una de dichas unidades temáticas. *Va a comenzar el programa deportivo.* || 11. Cada una de las operaciones que, en un orden determinado, ejecutan ciertas máquinas. || 12. *Inform.* Conjunto unitario de instrucciones que permite a un ordenador realizar funciones diversas, como el tratamiento de textos, el diseño de gráficos, la resolución de problemas matemáticos, el manejo de bancos de datos, etc. || 13. coloq. *Ecuad.* y *Ur.* Relación amorosa furtiva y pasajera. || 14. coloq. *Ur.* Cita amorosa. || 15. coloq. *Ur.* Persona con quien se tienen relaciones sexuales pasajeras. || ~ continuo. m. sesión continua.

Si bien la acepción 12 es la que corresponde a Informática, otras acepciones pueden ser útiles para entender mejor qué es un programa. Véase, por ejemplo, las acepciones 7, 8, 11.

Entendemos fácilmente cuando alguien habla de *programar* un viaje, *programar* una fiesta, etc. Se hace un programa, se programa una serie de actividades y una lista de las cosas que serán necesarias para lograr un objetivo dado. Así, si lo que se programa es una fiesta, será necesario "pensar" en las bebidas, en la comida, vajilla, etc. y en las actividades, acciones para obtener esas cosas, así como en las acciones a realizar con ellas.

En todos los casos hay un conjunto de elementos a tener en cuenta para llevar a buen puerto la actividad que se desea programar:

(a) Las *acciones* a realizar. Algunas veces en un orden determinado, una después de otra o inclusive algunas que pueden desarrollarse simultáneamente.

(b) Los *objetos* sobre los cuales se ejecutarán las acciones. Por ejemplo, en el caso de programar una fiesta una

acción será la de invitar a un grupo de personas (en este caso el objeto sobre el cual se ejecuta la acción es el conjunto de personas);

(c) Algo que es tan obvio que algunas veces se pasa por alto: el *ejecutor*, es decir, quien ejecutará, quien llevará a la práctica el programa.

Es de hacer notar que cualquiera sea la actividad que uno esté programando (por ejemplo, organizar un viaje, hacer una torta, etc.) lo primero que hacemos, o preparamos, es un programa de la misma antes de que el mismo sea puesto en ejecución. Hay también que pensar que hay programas y programas: algunos más complejos y/o largos que otros y que necesitan una mejor y más afinada preparación.

Es así como podemos distinguir dos etapas en cualquier actividad programable:

(1) el diseño, preparación, planificación, o creación del programa en sí mismo;

(2) la ejecución o puesta en práctica del programa.

Muchas veces, cuando creamos un programa, lo hacemos sólo en nuestra mente, es decir sin necesidad de anotar nada. Esto ocurre, generalmente, en aquellos casos en que las tareas a llevar a cabo son pocas y sencillas, y en consecuencia no se corre el riesgo de olvidar ningún paso. Esto presupone, por supuesto, que quien va a llevar a la práctica el programa, es decir, lo va a ejecutar, es la misma persona que lo diseñó.

Por el contrario, si el programa es largo, complejo y/o quien o quienes deberán llevarlo a la práctica no son los mismos que quienes lo han diseñado, entonces habrá que tomar nota paso a paso de todas las actividades que deben ser llevadas a cabo, es decir, escribir un programa, no sólo para no olvidar ninguna acción a realizar, ni objeto con el que haya que trabajar, sino, sobre todo, por si resulta necesario darle a alguien las instrucciones para llevar a cabo la actividad para la cual se ha hecho el programa.

Al dar las instrucciones habrá que ser extremadamente cuidadoso. En efecto, si uno espera que su programa se lleve a cabo con éxito, debe asegurarse que quién o quiénes lo ejecuten entiendan perfectamente las instrucciones que se les dan.

En consecuencia, deberemos elegir el lenguaje en que daremos las órdenes, así como las palabras empleadas, evitando usar un lenguaje desconocido para quien ejecutará el programa, así como palabras o giros idiomáticos incomprensibles para dichas personas. Es decir que estaremos obligados a usar un lenguaje que la o las personas, que ejecutarán nuestro programa, entiendan perfectamente.

Obviamente, el programa que hagamos no deberá tener ningún tipo de error si queremos que sea un éxito; ni errores en su redacción, que puedan causar que el mismo no sea comprendido por quienes deben ejecutarlo; ni errores en su concepción, que impliquen que aún cuando el programa se ejecute correctamente, es decir de acuerdo con las instrucciones dadas, el resultado no sea el esperado sino otro muy distinto o incluso que debido a los errores no pueda ser llevado a la práctica.

En consecuencia hay dos aspectos importantes a tener en cuenta:

(1) El *lenguaje* que se empleará para escribir el programa.

(2) La *corrección* del programa.

Es importante no sólo elegir un lenguaje adecuado, que sea comprensible por quienes van a ejecutar el programa y que no permita ambigüedades, es decir, interpretaciones múltiples, sino también es importante el uso que hagamos de este lenguaje. Con esto queremos decir que, en nuestros programas, no deben existir errores de escritura (ortografía) ni de uso incorrecto de la gramática del lenguaje (sintaxis). Es decir, también es importante la *corrección del programa*. Esto significa, además, que el programa que hagamos, las actividades programadas, los objetos empleados, serán aquellos que nos llevarán al resultado esperado. Es decir que las acciones programadas y el orden en el cual las mismas deben ser llevadas a cabo son las *correctas* para lograr nuestro objetivo. Más de una

vez, una actividad planificada de antemano no resulta como se esperaba porque se comenten errores en la planificación de la misma, en la programación de ella ya que alguna actividad estaba de más, o faltaba una actividad o se hizo una actividad en lugar de otra, o no se usaron los objetos correctos y necesarios para la actividad programada, o se los emplearon de manera incorrecta, etc.

Por supuesto, los *errores* en los resultados, también pueden deberse a mala interpretación, o a una incorrecta ejecución de los encargados de llevar a la práctica (ejecutar) el programa.

Resumiendo, podemos decir que la actividad de programar tiene las siguientes características:

(1) Hay *acciones* a realizar, la mayor parte de las veces en un orden determinado.

(2) Hay *objetos o cosas* sobre los que las acciones se realizan.

(3) Existen uno o más *ejecutores* del programa.

(4) Hay *dos etapas* bien distinguidas en una actividad programable:

(a) La *programación,* que es la preparación o confección del programa. En esta etapa las acciones y los objetos se definen. Los objetos necesarios para realizar las acciones se representarán de alguna manera y a dicha representación abstracta le llamaremos *datos*.

(b) La *ejecución* del programa, que es la puesta en práctica del mismo. En esta etapa las *acciones* del programa se *ejecutan*.

(5) Es necesario un *lenguaje* para describir las acciones y los objetos del programa. Este lenguaje debe tener las siguientes características:

(a) Ser *entendible,* conocido, por el *ejecutor* del programa.

(b) *No permitir*, en lo posible, *ambigüedades* en su empleo.

(6) En la programación es necesario:

(a) *Evitar errores de concepción* o de la *lógica* del programa, es decir, errores que lleven a un resultado distinto del esperado, aún cuando el programa se ejecute correctamente.

(b) *Evitar errores de sintaxis* del programa, es decir, errores de uso incorrecto del lenguaje de programación, que impidan que el ejecutor del programa entienda el mismo.

## 2.2 PROGRAMACIÓN DE COMPUTADORAS

La programación de computadoras comparte todas las características antes enunciadas.

Un aspecto importante es, sin embargo, que quien ejecuta el programa es una máquina. En consecuencia el lenguaje empleado para programar debe ser el que la máquina entienda. El uso del lenguaje de programación, la atención a su sintaxis, debe ser mucho más cuidadosa, ya que la máquina, la computadora, es mucho más estricta y rigurosa que un ser humano y, por eso mismo, intolerante con los errores sintácticos y más dispuesta a la ejecución literal del programa.

Por otro, lado las computadoras no pueden entender, por lo menos por el momento, los lenguajes naturales que son los empleados por los seres humanos. “Entienden”, si ésta palabra puede ser aplicada a las máquinas, un lenguaje bastante más restringido, más simple, pero al mismo tiempo más estricto y riguroso en su sintaxis que los lenguajes empleados por el hombre.

Los lenguajes de programación empleados para las computadoras han evolucionado y siguen evolucionando, tratando de hacerlos más fáciles en su uso y al mismo tiempo incorporando principios y técnicas de programación que no solo hacen a ésta más fácil sino menos sujeta a errores.

En efecto, la atención a los detalles en la programación, y sobre todo la cantidad de acciones a describir en un programa para una computadora, hace necesario el uso de técnicas de programación orientadas a evitar programas con errores. En algunos casos, estas técnicas se incorporan a los lenguajes de programación, en otros sólo son reglas empíricas o heurísticas a seguir en la confección de programas para evitar los errores.

## 2.3 PROGRAMACIÓN Y RESOLUCIÓN DE PROBLEMAS

Muchas veces, se conoce cómo resolver un problema y obtener el resultado deseado. Es decir, conocemos el *algoritmo* que resuelve el problema. Un *algoritmo*[1] es un conjunto pre escrito de instrucciones o reglas bien definidas, ordenadas y finitas que permite realizar una actividad mediante pasos sucesivos que no generen dudas a quien deba realizar dicha actividad. Dados un estado inicial y una entrada, siguiendo los pasos sucesivos se llega a un estado final y se obtiene una solución.

Cuando conocemos el algoritmo, se sabe entonces cómo programar una serie de actividades, con el objetivo de alcanzar el resultado. Es decir, conocemos el algoritmo que resuelve el problema y sólo debemos repetir sus pasos y acciones, empleando los objetos indicados para conseguir el resultado deseado. En estos casos, el problema ya está resuelto de antemano y solo hay que echar mano a la *solución conocida*, es decir, hay que llevar a la práctica, ejecutar, dicha solución o algoritmo.

Otras veces, si bien se conoce el problema que se desea resolver, no se conoce la forma de alcanzar la solución. Los pasos y acciones así como los objetos necesarios para obtenerlo, son desconocidos total o parcialmente. En estos casos, el problema no está resuelto y por el contrario debe encontrarse una solución para el mismo.

En el primer caso existe un algoritmo conocido que resuelve el problema y solo hay que aplicarlo; es decir seguir, ejecutar, los pasos indicados en el algoritmo o programa que lo implemente. En el segundo caso el algoritmo no existe. No hay una solución conocida al problema planteado para obtener el resultado esperado y la solución debe descubrirse. Hay que resolver el problema, encontrar la secuencia de pasos, acciones y las cosas y objetos necesarios, para obtener el algoritmo que nos permita alcanzar el resultado.

En general reservaremos el término *algoritmo* para una secuencia de instrucciones u órdenes encaminadas a dar la solución de un problema, escritas en un lenguaje no ambiguo *cercano* al de los *seres humanos*.

La palabra *programa* será la secuencia de instrucciones u órdenes encaminadas a dar la solución de un problema, escritas en un lenguaje *cercano* a las *máquinas*, generalmente llamado *lenguaje de programación*.

La programación está íntimamente ligada a la *resolución de problemas*. En caso que el algoritmo exista, solo se debe seguir sus indicaciones para obtener el resultado deseado. Por ejemplo, se conoce el algoritmo (ingredientes y pasos necesarios) para obtener ácido acetilsalicílico (la vulgar aspirina), solo hay que seguir dicha "receta" o algoritmo para fabricarlo. Se conocen los pasos necesarios para demostrar que existe una solución para la ecuación $x^2 + y^2 = z^2$ (Teorema de Pitágoras).

Pero se desconoce, lamentablemente hasta el momento, una cura definitiva contra el SIDA. No existe un algoritmo conocido que nos permita producirla. Este es un *problema que debe resolverse*, es un algoritmo que debe descubrirse si se quiere alcanzar el resultado deseado de una cura del SIDA. Asimismo, se desconocen los pasos necesarios para demostrar que si $n$ es un entero mayor que 2 no hay solución posible en los enteros positivos a la ecuación $x^n + y^n = z^n$; aún cuando Pierre de Fermat, en el siglo XVII dijo haber encontrado la solución pero nunca se encontró nada escrito. Yoichi Miyaoka, en 1988, parecía haberla encontrado, pero su demostración tenía un error.

[1] Definición tomada de Wikipedia (http://es.wikipedia.org/wiki/Algoritmo).

Andrew Wiles, en 1993, parece haber encontrado la solución. ¿Tendrá un error su solución?

Lamentablemente (o quizás deba decirse ¡por suerte!, quién sabe) no existe un algoritmo conocido que nos permita resolver problemas, cualquier problema. La resolución de problemas es un ámbito reservado a la inteligencia y habilidad de los seres vivos. Es, se podría decir, un arte. Existen, sin embargo, métodos y técnicas que pueden ayudar a resolver problemas. Algunas de orden muy general otras de aplicación a casos particulares.

La programación tiene como objetivo fundamental expresar en un lenguaje entendible por la computadora un algoritmo o solución a un problema dado, es decir, producir un programa. Muchas veces, nos enfrentaremos con problemas inéditos, es decir problemas cuya solución se desconoce. Otras veces, la solución es relativamente simple y puede obtenerse aplicando, por analogía, soluciones de problemas similares. Algunas veces basta con juntar las soluciones de varios problemas, ya resueltos, para obtener una nueva solución.

## 2.4 EL PROCESO DE RESOLUCIÓN DE PROBLEMAS

### INTRODUCCIÓN

Si programar implica resolver problemas, veamos un poco más de cerca qué significa esto.

Un *problema*, dice nuestro viejo amigo, el Diccionario Manual e Ilustrado de la Real Academia de la Lengua Española, es una "cuestión que se trata de aclarar". Es por eso que un problema lleva implícita la necesidad de resolverlo o aclararlo. Pero, a su vez, eso es justamente un problema: cómo resolver problemas, cómo aclararlos. Hasta ahora no existe una única e infalible *metodología de resolución de problemas*. No hay recetas que puedan aprenderse y que siguiéndolas nos lleven a la segura solución de cualquier problema. Hay, sin embargo, algunos *métodos generales* que, para algunos problemas, pueden ayudar en la resolución de los mismos.

Algunas veces es difícil, no ya la solución del problema sino, entender el mismo. Por ejemplo, mal puede intentarse encontrar la solución al problema de una cura contra el SIDA si desconocemos los términos en los cuales dicha cura debe basarse. Ni siquiera podemos plantear el problema (salvo en términos muy generales, tal como lo hemos hecho aquí), es decir desconocemos la naturaleza del mismo; o por ejemplo, ¿porqué con tres pedazos de madera de 6 cm., 8 cm. y 10 cm. de largo cada uno, se puede construir una escuadra? Para demostrar esto hay que poder definir el problema en términos más precisos, hay que poder conocer más del problema en sí.

Muchas veces describir claramente el problema, entender de qué se trata, puede ser el principio de su solución, sino la solución misma.

Es difícil pensar en resolver problemas, al menos problemas medianamente complejos, sin plantearlos antes y mucho más difícil concebir que puedan plantearse problemas complejos y resolverlos, encontrar una solución a los mismos, sin recurrir a algún tipo de lenguaje para registrar tanto el planteo del problema como su solución.

Esto equivaldría a preguntarse si podemos pensar sin tener algún lenguaje en el cual hacerlo. Este lenguaje puede tener muchas formas: hablado (sonido), escrito (con símbolos que equivalen a letras o palabras), puramente gráfico (con dibujos o diagramas); puede ser tan formal como el lenguaje matemático o tan informal como el lenguaje natural, pero parece imposible separar la actividad de pensar de la existencia de un lenguaje cualquiera.

De ahí que la actividad de resolver problemas se encuentre íntimamente relacionada con el lenguaje empleado para plantear y resolver los problemas. Tan relacionada está que algunas veces se confunde la tarea de resolver el problema con la de escribir su solución en un lenguaje cualquiera. Como los lenguajes tienen una sintaxis, es decir, tienen reglas para escribir correctamente, no debe confundirse el conocimiento de la sintaxis, y su correcto empleo, con una metodología de resolución de problemas.

Por otro lado, debemos convenir que existen problemas de todo tipo. Desde los más sencillos y cotidianos a los más complejos y difíciles. Así podemos agrupar los problemas, clasificarlos, y decir que una persona enfrentada con

un problema puede verse en las siguientes situaciones:

(a) Conoce no sólo el problema sino su solución. En este caso, sólo queda aplicar la solución para resolverlo.

(b) Conoce el problema y sabe que existe una solución aún cuando no la conozca, pudiendo recurrir a otra persona, o a un libro, que sí la conoce.

(c) Conoce el problema pero no la solución y sabe que no existe nadie ni nada que la tenga. En este caso, debe descubrir la solución. Es aquí donde podemos hablar de métodos de resolución de problemas, aún cuando el proceso es fundamentalmente individual y propio de la capacidad para resolver problemas de cada uno. En este caso, sin embargo, podemos separar algo más las características de los problemas:

(c.1) Cuando puede aplicarse una solución equivalente de un problema similar al que debe resolverse.

(c.2) Cuando existe una solución parcial, conocida, al problema.

(c.3) Cuando la solución del problema requiere la aplicación de varias soluciones conocidas, las que unidas lo resuelven.

(c.4) Cuando no se conoce una solución al problema y se sabe que no existe, por el momento, una solución al mismo, lo que no significa que no pueda encontrarse una. En este caso, una solución posible del problema es demostrar que el problema no tiene solución.

En computación y más concretamente, en programación, la mayor parte de los problemas más comunes y corrientes, se encuentran en los casos de los puntos (a) y (b); es decir que solo queda aplicar la solución, receta o algoritmo conocido. Hay muchos que entran en los grupos (c.1) a (c.3) y unos cuantos cuya solución se desconoce totalmente y que se encontrarían en el grupo (c.4).

Sin embargo, para alguien que empieza a aprender a programar es bueno considerar, hasta los problemas más sencillos, como pertenecientes a los del grupo (c.4).

En general, en programación, la solución de un problema (y su explicación, así como su enunciación) es el programa escrito en algún lenguaje de programación. Así se entiende que programar la solución a un problema, para que una computadora ejecute el programa y resuelva el problema (si el programa no tiene errores y es la solución correcta), consiste en escribir, en *codificar*, la solución encontrada, usando un lenguaje de programación.

De esto surge un texto con el programa escrito en el lenguaje de programación elegido, el que debe ser incorporado a la computadora para que esta someta dicho texto de programa a un *compilador*, el cual es a su vez un programa que, al ejecutarse en la computadora, resuelve el problema que significa traducir del lenguaje de programación, en el cual el programa ha sido escrito, al lenguaje que pueda entender la CPU, el *lenguaje de máquina*.

## ETAPAS DE LA RESOLUCIÓN DE PROBLEMAS

Es difícil, como ya se ha dicho, dar reglas generales de resolución de problemas. Hay muchos autores que se han dedicado a proponer métodos de resolución de problemas, particularmente en el área de la programación de computadoras.

Un principio ampliamente difundido es el de "*dividir para reinar*". Enfrentado con un problema complejo, y cuya solución no se encuentra fácilmente, es conveniente separar el problema en partes, en problemas más chicos y por eso mismo más manejables y para los que, quizás, ya exista una solución.

Por supuesto este principio no tiene limitación alguna. Puede dividirse un problema, en tantos problemas menores, como se desee. Asimismo, cada uno de estos problemas puede, a su vez, dividirse cuantas veces se desee o sea conveniente.

Ahora cada parte del problema general y más grande, puede ser atacada por separado y quizás entendida y resuelta más fácilmente. Solucionada cada una de las partes es probable que se tenga la solución del problema original. Es probable pero no seguro: la unión de cada una de las partes para conformar la solución general no siempre es tan fácil como parece.

Cómo un ejemplo posible vamos a establecer una serie de pasos o etapas que parece razonable tener en cuenta cuando se intenta resolver un problema complejo, o cada una de las partes de un problema, y cuya solución se desconoce inicialmente.

Estas etapas son solo a título de ejemplo y es posible que haya otras mejores y más efectivas. Recuérdese que al fin de cuentas la resolución de problemas depende más de quién lo resuelve que del método empleado para ello.

1.- *Definir* el problema (objetivo).

Parece razonable que, antes de intentar resolver un problema, se sepa de qué se trata, es decir cuál es el problema a resolver. Para ello es útil definir el problema:

1.1.- Primero, en términos sencillos y generales, empleando un lenguaje común y generalmente natural.

1.2.- Segundo, tratando de avanzar en una definición en términos lo más formales posibles.

2.- Determinar la *naturaleza* del problema.

Es bueno intentar saber si el problema es conocido, si ya tiene una solución conocida, si tiene una solución compuesta de soluciones ya conocidas.

3.- Determinar las *salidas* del problema resuelto.

En el caso de la programación de computadoras es útil definir las salidas, es decir, los resultados que el problema resuelto producirá. Definir qué datos, de qué tipos serán los resultados y dar algunos ejemplos de los resultados que se obtendrán.

4.- Determinar las *entradas* necesarias.

Si se conocen las salidas esperadas (los resultados del problema resuelto) habrá que definir las entradas necesarias (los datos, y sus tipos) para obtener las salidas del problema resuelto, dando algunos ejemplos de las mismas para los resultados esperados.

5.- *Diseñar* la solución.

Es útil comenzar a bosquejar una solución tentativa del problema, empleando en lo posible lo obtenido en los pasos anteriores, especialmente en los puntos 1 y 2. Por otro lado, este bosquejo inicial de la solución puede ser tratado en dos partes:

5.1.- Describir, inicialmente, informalmente la solución en términos de acciones generales, empleando el principio “dividir para reinar”.

5.2.- Describir, luego, formalmente la solución, obtenida en 5.1 en términos de algún lenguaje algorítmico.

6.- *Diseñar* la solución de cada *trozo* del problema.

Para cada pedazo del problema, que pueda haberse separado, debe aplicarse el mismo criterio que para el problema general, hasta encontrar la solución del mismo. El empleo sistemático de los pasos 1 a 6 para cada trozo de solución constituye el principio de *desagregación*: ir de lo más general a lo particular.

7.- *Ejecutar* el algoritmo *manualmente*.

Es sumamente conveniente convencerse que la solución encontrada es la correcta. Para ello un mecanismo posible consiste en ejecutar manualmente el algoritmo de la solución, por medio de ejemplos concretos. Esto es

una forma más de comprobar que la solución hallada es la correcta.

8.- *Codificar* la solución en el lenguaje de programación elegido.

Sólo al final, una vez encontrada la solución, debe codificarse ésta en un lenguaje de programación. Es poco conveniente atarse a los problemas de sintaxis antes de tener una solución al problema, ya que éstos pueden oscurecer o dificultar la búsqueda de la solución del problema. La solución de los problemas de codificación, de los problemas sintácticos, debe dejarse para el final, ya que son los que más fácilmente pueden resolverse.

## 2.5 LENGUAJES DE PROGRAMACIÓN: LENGUAJES DE MÁQUINA, LENGUAJES ENSAMBLADORES Y LENGUAJES DE ALTO NIVEL

Como dijimos antes, para escribir un programa de computadora debemos hacer uso de un lenguaje que la computadora pueda entender, es decir, un *lenguaje de programación*.

Existen distintos tipos de lenguajes de programación, algunos pueden ser comprendidos en forma directa por la máquina, mientras que otros necesitan de un proceso intermedio de traducción al lenguaje que la máquina entiende. Así es que los lenguajes de programación pueden clasificarse en tres grandes grupos: *lenguajes de máquina*, *lenguajes ensambladores* y *lenguajes de alto nivel*.

Un *lenguaje de máquina*, como su nombre lo indica, es el "lenguaje natural" de una máquina particular. Cualquier computadora puede comprender en forma directa solamente su propio lenguaje, el cual se encuentra definido por el diseño del hardware de esa computadora.

Los lenguajes de máquina, generalmente, consisten en cadenas de números, las que a la larga son reducidas a cadenas de 0's y 1's. Estas cadenas instruyen a la computadora para que lleve a cabo, de a una a la vez, operaciones elementales (operaciones que esa máquina puede realizar). Cada tipo de computadora particular tiene su propio lenguaje, es decir, los lenguajes de máquina son dependientes de la máquina, por lo tanto un lenguaje de maquina particular puede ser usado sólo en un tipo de computadora.

Los lenguajes de máquina resultan oscuros y difíciles de leer a las personas. Esto, a su vez, dificulta la tarea de programar y depurar los programas con ellos escritos. Es por esto que, históricamente, a medida que los programas fueron creciendo en tamaño, surgió la necesidad de usar abreviaturas en lenguaje natural para representar esas cadenas de 0's y 1's que la máquina podía comprender pero que a las personas resultaban crípticas, surgiendo así los primeros *lenguajes de ensamblado* (*assembly languages*, en inglés) junto con sus respectivos programas traductores, llamados *ensambladores* (*assemblers*, en inglés), que permiten traducir del lenguaje ensamblador al lenguaje de máquina.

Si bien esto facilitó en gran medida la tarea de programar, aún resultaba una tarea lenta ya que, para llevar a cabo hasta la tarea más simple, se requería de un gran número de instrucciones del lenguaje de ensamblado. Es así que surgieron entonces los lenguajes de programación conocidos como *lenguajes de alto nivel*, donde cada sentencia del lenguaje involucra un grupo de instrucciones de máquina.

En contraposición a los lenguajes de alto nivel, se suele hacer referencia a los lenguajes de máquina como lenguajes de bajo nivel. Los lenguajes de alto nivel se encuentran más lejos de lo que la máquina comprende y más cerca del programador. Acompañando a los lenguajes de alto nivel, surgieron los respectivos programas traductores conocidos con el nombre de *compiladores*, los que convierten del lenguaje de alto nivel al lenguaje de máquina. Estos lenguajes tienen una *sintaxis* que los define. La sintaxis es un conjunto de reglas que especifican cuáles son los elementos constitutivos válidos del lenguaje y de qué manera estos pueden combinarse entre sí para escribir programas sintácticamente correctos. Los programas sintácticamente correctos son los que pueden ser traducidos por el compilador al lenguaje de máquina.

Una ventaja importante de los lenguajes de alto nivel es la *portabilidad*. Esto significa que los programas

escritos en un lenguaje de alto nivel, almacenados en archivos fuente, pueden ser portados a distintas computadoras, con distintos sistemas operativos, siempre que contemos con el compilador para esa máquina. Algunas veces la compatibilidad no es total y se hacen necesarias algunas modificaciones menores en el programa fuente. Para evitar esto, muchos lenguajes de programación son estandarizados por comités del ANSI (American National Standard Institute) con el objeto de garantizar su portabilidad entre distintas computadoras.

Existen un gran número de lenguajes de alto nivel. En este curso vamos a usar el lenguaje C, un lenguaje imperativo ampliamente empleado, que si bien cuenta con las estructuras típicas de los lenguajes de alto nivel cuenta también con muchas características de bajo nivel.

## 2.6 LOS PARADIGMAS DE LA PROGRAMACIÓN

Un *paradigma* es un ejemplo o algo ejemplar. En Computación, la palabra paradigma se emplea en varios contextos. Hablamos de *paradigma de programación* para designar un conjunto de reglas, métodos y principios que comparten una filosofía común de programación.

En general, si bien algunos autores dan otras clasificaciones, los principales paradigmas de programación son el *imperativo*, el *funcional*, el *lógico* y el *orientado a objeto*. Sobre el paradigma orientado a objetos puede discutirse si es un paradigma como lo son los otros tres anteriores, o si es sólo un método que puede aplicarse en cualquiera de los otros paradigmas.

El *paradigma imperativo* se basa, fundamentalmente, en programar dando órdenes o instrucciones en forma imperativa, como su nombre lo indica. En él se describe la solución a un problema dando un conjunto de instrucciones que deben ejecutarse paso a paso una después de la otra, las cuales van cambiando el estado del programa. Algunos ejemplos de lenguajes imperativos son COBOL, FORTRAN, BASIC, Pascal, C, Ada, entre otros.

El *paradigma funcional* describe la solución del problema a través de la aplicación de funciones que lo resuelven. Los lenguajes funcionales, aún cuando el paradigma existe desde hace tiempo, no han tenido tanta difusión. Algunos ejemplos son Lisp, SNOBOL, Haskell, Miranda, Scheme, entre otros.

El *paradigma lógico* emplea la lógica matemática para dar la solución por medio de una descripción de relaciones lógicas entre los objetos que forman parte del problema. PROLOG es el lenguaje más conocido del paradigma lógico y cuenta con diversas variantes.

El *paradigma orientado a objeto* considera cada uno de los objetos que forman parte del problema así como aquellos que forman parte de la solución, junto a las acciones que sobre cada uno de estos objetos se realizan, como un todo único e inseparable, permitiendo la manipulación de los objetos sólo por medio de acciones previamente definidas para ellos. SMALLTALK es tradicionalmente el lenguaje asociado al paradigma de objetos, aún cuando se han hechos muy populares las extensiones, con principios de la metodología de objetos, a los lenguajes imperativos, tales como Object Pascal, C++, etc.

Cada paradigma ha sido llevado a la práctica, implementado, en uno o varios lenguajes de programación. Muchas veces, sin embargo, estos lenguajes no son ‘puros’, es decir que contienen principios propios de más de un paradigma. Así, por ejemplo, si bien C++ o Java son considerados lenguajes orientados a objetos, no son puros como lo es SMALLTALK. Tanto en Java como en C++ existe una diferencia entre valores que son objetos y valores que son de tipos primitivos, mientras que en SMALLTALK todos los valores son objetos. Incluso en C++, dado que se trata de una extensión al lenguaje C, podemos escribir programas que sean puramente imperativos. Asimismo, muchos lenguajes imperativos incluyen principios (a través de estructuras sintácticas del lenguaje) del paradigma funcional.

## 2.7 EL PARADIGMA DE PROGRAMACIÓN IMPERATIVA

La *programación imperativa* es un estilo o paradigma de programación en donde se describe la computación en términos de sentencias u órdenes que cambian el estado de un programa.

En la programación imperativa, el programa prescribe, en término de secuencias de acciones, cómo hacer para alcanzar la solución a un problema. Esto puede verse como opuesto a la programación declarativa (por ejemplo, la programación lógica y la programación funcional) que expresa lo que el programa debe hacer y no cómo hacerlo.

Los primeros lenguajes imperativos fueron los lenguajes de máquina. En estos lenguajes, las instrucciones son muy simples, lo cual dificulta la creación de programas complejos. FORTRAN, un lenguaje imperativo de alto nivel, fue el primer lenguaje de programación en superar los obstáculos presentados por el código de máquina en la creación de programas complejos.

La programación imperativa, o por *frases*, consiste justamente en describir la solución de un problema a través de las *órdenes*, *frases*, *instrucciones* o *sentencias*, necesarias para resolverlo. Estas órdenes describen las acciones que deben realizarse sobre los *datos* que forman parte del problema. Así, un programa imperativo es un conjunto de sentencias que le indican a la computadora cómo realizar una tarea.

En consecuencia, en un programa, distinguimos dos grandes partes:

(a) los *datos* que forman parte del problema o que se requieren para resolverlo.

(b) las *acciones* necesarias para resolver el problema, y que procesan los datos.

## 2.8 ETAPAS EN LA CONSTRUCCIÓN DE UN PROGRAMA

La construcción de un programa sencillo, como los que desarrollaremos en este curso, típicamente pasa a través de una serie de etapas, como puede verse en la Figura 2.1:

1. *Diseño de la solución*, empleando algún método y diversas herramientas: diagramas, lenguaje natural (el castellano, por ejemplo), algún lenguaje propio del ámbito del problema a resolver (el lenguaje de la Matemática, por ejemplo, si el problema a resolver es de ese ámbito), etc.
2. *Codificación de la solución* en algún lenguaje de programación. Muchas veces, esta etapa es llevada a cabo en forma conjunta con la etapa 3 (*Edición*).
3. *Edición del programa*, que es la escritura del programa sobre la computadora. Esto es llevado a cabo con un programa, llamado *editor*. La mayor parte de los ambientes de programación de hoy en día tienen integrados el programa editor. Uno puede escribir el programa, hacer modificaciones y guardarlo (grabarlo) en algún dispositivo de almacenamiento secundario, tal como un disco rígido. El programa, escrito en un lenguaje de programación de alto nivel, se suele almacenar en uno o más archivos llamados *archivos fuente*.
4. *Compilación del programa*, corresponde a la etapa de traducción del programa escrito en el lenguaje de programación de alto nivel elegido al lenguaje que “entiende” la máquina en donde se ejecutará el programa. La compilación es llevada a cabo por otro programa particular: el *compilador* del lenguaje de programación elegido.

   En esta etapa, cuando ejecutamos el compilador del lenguaje, debemos pasarle como entrada nuestro programa, escrito en ese lenguaje de programación (por ej. Pascal, C, Java, etc.), es decir, el nombre del archivo o de los archivos fuente que contienen el programa.

   Una vez que el compilador se ejecuta sobre el programa fuente, producirá como salida una nueva versión de nuestro programa ahora escrita en el lenguaje que entiende la CPU en la cual estemos trabajando. La compilación del programa se suele descomponer en dos etapas, que se pueden realizar juntas o por separado. En

primer lugar el compilador traduce el programa fuente a lenguaje ensamblador, produciendo lo que se conoce como *archivo objeto*. En una segunda etapa se lleva a cabo el proceso de *enlazado* (o *linking*) del archivo objeto, produciendo un *archivo ejecutable* que contiene el programa traducido al lenguaje de máquina.

Si durante la etapa de compilación, el compilador detectase que hay errores de sintaxis en nuestro programa, deberemos volver a editar el programa (paso 2) para corregir los errores de sintaxis detectados.

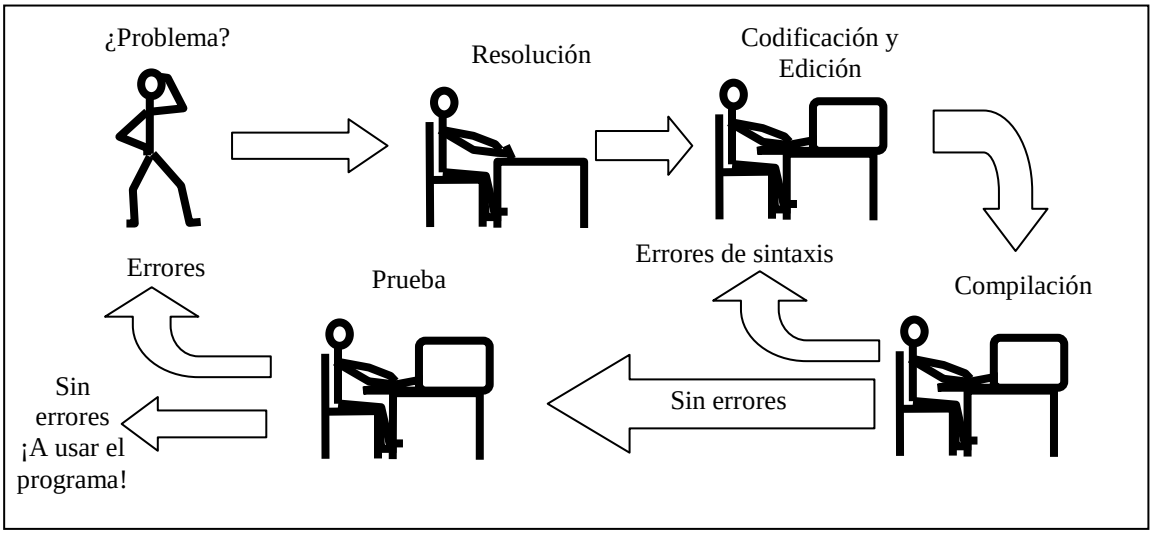


**Figura 2.1. El proceso de resolución de problemas e implementación de un programa que lo resuelve.**

5. *Ejecución* del programa, si no hay errores de sintaxis detectados durante la compilación. Durante esta etapa el archivo ejecutable producido por el compilador es cargado por el sistema operativo en la memoria RAM de la computadora y se ejecuta.

Por otro lado, si hubo errores en la solución diseñada en el paso 1 y se detectaron en la etapa de ejecución del programa (paso 5), será necesario volver al paso 1 (diseño de la solución), repitiendo nuevamente todos los pasos cuantas veces sea necesario hasta que el programa funcione correctamente.

# 3 PROGRAMACIÓN EN C

## 3.1 INTRODUCCIÓN

El lenguaje C fue desarrollado por Dennis Ritchie en los Laboratorios Bell entre los años 1969 y 1972. Su nombre se debe a que fue pensado como una evolución del lenguaje B, creado por Ken Thompson también en los Laboratorios Bell.

C contiene muchos conceptos importantes de B y de BCPL, el lenguaje que dio origen a B, a la vez que agrega nuevas características.

La fama de C nace por ser el lenguaje usado para el desarrollo del sistema operativo UNIX. Hoy en día, la mayor parte de los sistemas operativos se encuentran escritos en C y/o C++.

Existen compiladores de C para la mayoría de las computadoras, a la vez que es posible escribir programas en C que pueden ser portados a otras computadoras. En efecto, existe un C estándar, el ANSI C, que es la evolución del C de Kernighan y Ritchie (también conocido como el "C clásico o tradicional") creado con el objetivo de unificar la gran cantidad de variantes del lenguaje que fueron surgiendo a lo largo del tiempo. La segunda edición del libro de Kernighan y Ritchie, publicada en 1988, cubre al estándar ANSI C.

## 3.2 LA ESTRUCTURA GENERAL DE UN PROGRAMA EN C

Un programa sintácticamente correcto en C, cualquiera sea su tamaño, consiste de *funciones*. En particular, una función que nunca puede faltar es la función de nombre `main`, la cual corresponde al *programa principal* por donde comienza la ejecución del programa. Por lo general, `main` llamará a otras funciones que le ayudarán a realizar su cometido. Algunas de estas funciones las escribiremos nosotros, otras corresponderán a funciones previamente escritas, que ya se encuentran disponibles en *bibliotecas*.

Podemos comenzar escribiendo el típico programa que muestra por pantalla el mensaje "Hola mundo!":

```
/* Mi primer programa C */

#include <stdio.h>

main()
{
      printf("Hola mundo!\n");
}
```

Lo primero que podemos notar es la línea `/* Mi primer programa C */`, que corresponde a un *comentario* en C. Los comentarios sirven para documentar el programa y facilitar su lectura, pero no provocan la ejecución de ninguna acción. Los comentarios en C, comienzan con `/*` y terminan con `*/`.

La línea `#include <stdio.h>` indica al compilador que debe incluir el contenido del archivo `stdio.h`, el cual contiene la biblioteca de funciones de entrada/salida estándar. Este archivo debe ser incluido en cualquier programa C que produzca salidas por pantalla o entradas de datos con el teclado.

Una forma de pasarse datos entre las funciones del programa es por medio de una lista de valores llamados *argumentos o parámetros*. Los paréntesis, que van después del nombre de la función, encierran la lista de parámetros. Los paréntesis `()` después de `main` indican, en este caso, que la función `main` no tiene parámetros.

Entre las llaves `{}` escribiremos todas las sentencias necesarias para que nuestro programa haga lo que deseemos. Las llaves indican que el grupo de sentencias encerradas corresponden a un *bloque de sentencias* o

*sentencia compuesta*.

La línea `printf("hello, world\n!");` es una *sentencia de invocación* a la función `printf` de la biblioteca de funciones estándar `stdio.h`; esta función le indica a la computadora que debe mostrar por la pantalla la cadena de caracteres que se encuentra entre las comillas dobles (`""`). Toda sentencia en C debe terminar con un punto y coma (`;`).

Si ejecutamos el programa, notaremos que los caracteres `\n` no son visualizados como tales en la pantalla. En C, el carácter *backslash* (\) es un carácter especial llamado *carácter de escape*. Este le indica a la instrucción `printf` el comienzo de una *secuencia de escape*. En particular, `\n` es la secuencia de escape correspondiente a una nueva línea (*newline*), que hace que el cursor se posicione en la línea de abajo. En la Tabla 3. 1 se muestran las secuencias de escape más usadas en C.

**Tabla 3. 1. Principales secuencias de escape en C**

| Secuencia de escape | Descripción |
|---|---|
| `\a` | Carácter de alarma (campana del sistema). |
| `\n` | New line (Nueva línea). Posiciona el cursor de la pantalla al comienzo de la próxima línea. |
| `\r` | Carriage return (Retroceso de carro). Posiciona el cursor al comienzo de la línea corriente sin avanzar a la próxima. |
| `\t` | Tabulador horizontal. Mueve el cursor hasta la próxima marca de tabulación. |
| `\\` | Backslash. Usado para imprimir el carácter backslash. |
| `\"` | Comilla. Usado para imprimir la comilla. |
| `\'` | Apóstrofo. Usado para imprimir el apóstrofo. |
| `\?` | Signo de interrogación. Usado para imprimir el signo de interrogación. |

## 3.3 PROGRAMACIÓN Y DATOS

Un *dato* es la representación dentro de la computadora de algún aspecto de la realidad. Estos pueden para ser dados como entradas a un programa, procesados y devueltos como resultados.

Si los datos son la representación de aspectos de la realidad, y dado que hay distintos aspectos de la realidad, estos deben poder reflejar esa diversidad. Para esto existen los *tipos de datos.*

Un *tipo de dato* es un conjunto de valores que comparten las mismas características y operadores. Por ejemplo, un tipo de dato muy empleado y común en los lenguajes de programación es el tipo *entero* que representa a los números enteros, otro tipo de dato es el llamado tipo *carácter* que agrupa a los símbolos empleados en la escritura: las letras del abecedario, donde se distinguen las mayúsculas de las minúsculas, los símbolos que representan los números del cero al nueve y varios símbolos de puntuación (. , ; : ‘ “ ¡ etc.). Existen muchos otros tipos de datos y cada lenguaje de programación define los propios.

Un dato puede representarse en un programa por medio de una *constante simbólica* o *literal*, que es, como su nombre lo indica, un valor que se mantiene sin cambio alguno durante toda la ejecución del programa, es decir es invariable o constante. Por ejemplo, `114` es la constante que representa al número entero ciento catorce. Como dato

adicional piénsese que el número ciento catorce lo representamos con la constante `114`, que no es más que la concatenación de los símbolos `1`, `1`, `4`, que son los que representan los números uno, uno y cuatro. Sin embargo, `CXIV` ó `1110010` también son representaciones del mismo número `114`, aunque con otros símbolos.

Los datos, en un programa, también pueden representarse por medio de *variables* que son, como su nombre lo indica, elementos sintácticos que representan o contienen, en cada momento de la ejecución, uno de los posibles valores del tipo de dato de la variable.

Una variable tiene tres atributos: un *nombre* o *identificador*, un *tipo de dato* y un *valor*. El identificador es el nombre que le asignamos a la variable y que usamos para referirnos a ella dentro del programa, mientras que su tipo de dato determina el conjunto de todos los posibles valores que la variable puede tomar durante la ejecución del programa. En cada instante de la ejecución del programa, cada variable asumirá un único valor de todos los posibles valores. Ese valor podrá ir cambiando (variando) a lo largo de la ejecución.

En consecuencia, podemos visualizar una variable como un lugar de la memoria de la computadora, con un tipo de dato asociado y un nombre (dado por su identificador), en el que pueden almacenarse cualquiera de los valores del tipo de dato asociado a la variable. Por ejemplo, `peso` podría ser el nombre (el identificador) de una variable que puede almacenar valores de tipo real (números reales).

Es una buena técnica de programación *declarar* todos los datos que constituirán tanto la entrada como la salida, así como los intermedios, de un programa. Es así como muchos lenguajes de programación exigen estas definiciones y declaraciones en el programa que los procesará.

En general, un programa recibe datos de entrada, los que son procesados por la computadora y sobre ellos se pueden efectuar operaciones por medio de *operadores*. Los operadores, entonces, denotan las operaciones que pueden efectuarse sobre los datos. Por ejemplo, el operador + que, generalmente, en muchos lenguajes de programación denota la operación de suma de datos numéricos.

El ejemplo de la Figura 3.1 muestra un programa C que imprime por pantalla la suma de dos números enteros.

```
/* Programa de suma de dos enteros */

#include <stdio.h>

main()
{
    int entero1, entero2, suma;        /* declaraciones */

    printf("Ingrese el primer entero\n");
    scanf("%d", &entero1);             /* lee un entero en entero1*/
    printf("Ingrese el segundo entero\n");
    scanf("%d", &entero2);             /* lee un entero en entero2*/
    suma = entero1 + entero2;          /* asignación de la suma a suma */
    printf("La suma es %d\n", suma);   /* visualiza la suma */
}
```

**Figura 3.1 Programa que suma dos enteros**

Lo primero que podemos notar como novedoso en este programa es la línea `int entero1, entero2, suma` donde se *declaran* tres variables de nombre `entero1, entero2` y `suma` y cuyo tipo de dato es `int`. Esto significa que estas tres variables podrán contener valores enteros.

En C, se deben declarar todas las variables antes de su uso. Generalmente, esto se hace al principio de la función

antes de las sentencias ejecutables.

En C, el nombre de una variable puede ser cualquier identificador válido. Los *identificadores válidos* pueden contener letras, números y el carácter de subrayado (_) pero no pueden comenzar con un número; pueden tener cualquier largo, aunque el compilador sólo reconoce los 31 primeros caracteres. C es *case sensitive*, lo que significa que hace una diferencia entre letras mayúsculas y minúsculas, por lo tanto el identificador `Peso` es distinto del identificador `peso`. Tampoco pueden usarse *palabras claves* del lenguaje como identificadores, como por ejemplo, `if`, `else`, `int`, `float`, `while`, `do`, etc.

La línea `scanf("%d", &entero1);` es una sentencia de invocación a la función `scanf` que permite el ingreso de datos desde la entrada estándar (que usualmente es el teclado). Tiene dos argumentos o parámetros: `"%d"` es el *string de control de formato* de la entrada, que indica el tipo de dato que deberá ser ingresado por el usuario. En este caso, el string de control de formato `"%d"` indica que el dato deberá ser un entero. El segundo parámetro está formado por el *operador de dirección* & seguido por el nombre de una variable. Este parámetro indica la dirección de la variable donde se almacenará el dato entero que el usuario ingrese. El uso del operador *ampersand* (&) resulta un poco confuso, pero por ahora recordemos solamente que cuando usemos `scanf` deberemos anteponer & a los nombres de las variables en las cuales vayamos a almacenar los datos que se lean. Más adelante comprenderemos por qué esto es necesario y veremos también que hay algunas excepciones a esta regla.

La línea `sum = entero1 + entero2;` corresponde a una sentencia de asignación donde se calcula la suma de las variables `entero1` más `entero2` y lo almacena en la variable `sum`.

La instrucción `printf("La suma es %d\n", sum);` tiene dos parámetros. El primero es un *string de control del formato* de la salida. Este string contiene los caracteres `%d` que indican que en ese lugar será visualizado un número entero. El segundo argumento, `sum`, indica el valor entero a visualizar en el lugar de `%d`. El mismo efecto podría haberse logrado por medio de la sentencia `printf("La suma es %d\n", entero1 + entero2);`

## 3.4 TIPOS DE DATOS, OPERADORES Y EXPRESIONES EN C

La Tabla 3. 2 muestra una posible clasificación de los tipos de datos del lenguaje C.

**Tabla 3. 2. Tipos de datos en C**

<table>
<tr><td rowspan="6">Tipos de datos en C</td><td rowspan="4">Básicos</td><td rowspan="3">Aritméticos</td><td>Enteros</td></tr>
<tr><td>Caracteres</td></tr>
<tr><td>Flotantes o reales</td></tr>
<tr><td colspan="2"><code>void</code></td></tr>
<tr><td rowspan="3">Estructurados o Compuestos</td><td colspan="2">Arreglos</td></tr>
<tr><td colspan="2">Estructuras o registros</td></tr>
<tr><td colspan="2">Uniones</td></tr>
<tr><td colspan="3">Punteros</td></tr>
</table>

En la sección siguiente trataremos los tipos de datos aritméticos así como sus operadores. El tipo `void` es introducido junto con el concepto de función en C, cuando tratemos el tema de modularidad en el capítulo 4. Los punteros son tratados en el capítulo 5 y los tipos estructurados de C en el 7.

# TIPOS DE DATOS ARITMÉTICOS

## ENTEROS

Existen en C tres tamaños de enteros: `short int`, `int` y `long int`. Los objetos de datos de tipo `int` tienen el tamaño natural sugerido por la arquitectura de la máquina en donde se ejecuta el programa. Esto significa que si el tamaño del registro del procesador es de `16` bits, entonces se espera que los datos de tipo `int` tengan `16` bits; si es `32` entonces los `int` tendrán `32` bits, etc.

Los `long int` tienen al menos tantos bits como los `int` y los `short int` tienen a lo sumo tantos como los `int`. Sin embargo, esto dependerá de cada implementación, pudiendo ocurrir que los tres fuesen equivalentes.

Los tres especifican valores con signo a menos que se indique lo contrario. Para eso está el calificador `unsigned` que indica que los datos sólo pueden tomar valores positivos, ampliando de esta forma el rango de valores posibles. Así por ejemplo, si la palabra de la máquina tiene 16 bits, el rango de un `int` será `-32768 a 32767`, mientras que el de un dato de tipo `unsigned int` será de `0` a `64535`.

No es necesario usar la palabra clave `int` cuando aparece alguno de los otros modificadores. Así, por ejemplo, las siguientes declaraciones de variables

```
unsigned int x;
short int y;
unsigned long int xy;
```

podrían haberse escrito también así:

```
unsigned x;
short y;
unsigned long xy;
```

## CARACTERES

Las datos de tipo `char` (carácter) sirven para representar un único carácter. Ocupan un *byte* de memoria (8 bits). Ejemplos válidos de declaraciones son las siguientes:

```
/* declara variable x de tipo char */
char x;

/* declara en una sola línea dos variables de tipo char */
char w,y;

/* declara variable z de tipo char y le asigna el valor inicial 'A' */
char z = 'A';
```

En C no existe una diferencia entre los caracteres y los enteros. Un carácter es un caso particular de un entero de 1 byte. En consecuencia, la declaración anterior podría haberse escrito también

```
char z = 65;
```

puesto que el código ASCII del carácter `'A'` es `65`.

Asimismo, puesto que se trata de un entero, podemos realizar sobre los caracteres todas las operaciones definidas para los enteros. Por ejemplo,

```
z = z + 1;
printf("El código ASCII de %c es %d\n", z, z);
```

Luego de ejecutarse esas dos sentencias y dado que "%c" es el especificador de formato para caracteres, imprimirá

```
El código ASCII de B es 66
```

## FLOTANTES O REALES

Existen tres tipos de datos flotantes: `float` (punto flotante de precisión simple), `double` (punto flotante de precisión doble) y `long double` (punto flotante de precisión extra). Sin embargo, al igual que ocurre con los datos enteros, la precisión dependerá de la implementación, pudiendo los tres ser equivalentes o siendo los últimos de la lista al menos tan precisos como los que los preceden.

El programa mostrado en la Figura 3.2 usa datos de punto flotante para calcular el área de un círculo de radio dado.

```
/* Programa que calcula la superficie de un círculo de radio dado */
#include <stdio.h>

main()
{
  float radio;

  printf("Ingrese el radio del circulo en cm: ");
  scanf("%f", &radio); /* lee el radio*/
  printf("El circulo de radio %f tiene una superficie de %f cm2\n",
           radio, 3.14159265358979323846 * radio * radio);
}
```

**Figura 3.2. Programa que calcula la superficie de un círculo.**

Es una mala práctica de programación usar "números mágicos" en un programa, como por ejemplo, en este caso, el número `3.14159265358979323846`, ya que proporcionan poca información a quien deba leer el programa. Peor aún, dificultan la modificación del programa cuando el valor se encuentra repetido en muchos puntos del mismo. Una manera de evitar este inconveniente es atribuirle al valor un nombre significativo. Esto podemos hacerlo en C definiendo una *constante simbólica* por medio de una orden `#define` en donde la constante es definida como una cadena de caracteres. En la Figura 3.3 se muestra el programa del círculo usando una constante simbólica `PI`.

```
/* Programa que calcula la superficie de un círculo de radio dado */

#include <stdio.h>
#define PI  3.14159265358979323846     /*definicion de constante PI*/

main()
{
    float radio;

    printf("Ingrese el radio del circulo en cm: ");
    scanf("%f", &radio);
```

```
        printf("El circulo de radio %f tiene una superficie de %f cm2\n",
              radio, PI * radio * radio);
}
```

**Figura 3.3. Programa que usa constante simbólica `PI` para calcular la superficie de un círculo de radio dado.**

Los nombres de las constantes simbólicas siguen las mismas reglas que las de las variables: secuencias de letras, dígitos y _ que no comienzan con un número. Por convención se las escribe usando sólo letras mayúsculas.

La línea `#define PI 3.14159265358979323846` define la constante simbólica `PI`. A partir de esto, cualquier ocurrencia de `PI`, que no se encuentre entre comillas ni como parte de otra palabra en el programa, como por ejemplo la ocurrencia de `PI` en la línea `printf("El circulo de radio %f tiene una superficie de %f cm2\n", radio, PI * radio * radio);`

será sustituida por el preprocesador de C por `3.14159265358979323846`

El preprocesador de C es, como su nombre lo indica, un programa que se corre previo al proceso de compilación de un programa C y que lleva a cabo ciertas manipulaciones en ese programa fuente antes de su compilación. Estas manipulaciones consisten en resolver las directivas u órdenes que son para el preprocesador. Si bien hay otras, las directivas más comúnmente usadas son `#include` y `#define`. En el caso del `#include`, el preprocesador incluye en el archivo que se va a compilar, el archivo fuente indicado en el `#include`, mientras que el caso de la directiva `#define` realiza un reemplazo de texto.

## OPERADORES Y EXPRESIONES

### OPERADORES Y EXPRESIONES ARITMÉTICAS

Sobre los datos es posible llevar a cabo operaciones. En particular, para los datos aritméticos existen una serie de operadores aritméticos predefinidos que podemos combinar de diversas maneras obteniendo nuevas expresiones aritméticas.

**Tabla 3.3. Operadores aritméticos**

| Operador aritmético | Operador en C | Expresión aritmética | Expresión en C |
|---|---|---|---|
| Suma | + | $x + 20$ | `x + 20` |
| Resta | - | $a - b$ | `a - b` |
| Multiplicación | * | $xy$ | `x * y` |
| División | / | $x/y$<br>$x:y$<br>$\frac{x}{y}$ | `x / y` |
| Módulo | % | $u \bmod k$ | `u % k` |

La Tabla 3.3 muestra los operadores aritméticos binarios, es decir, que toman dos operandos. Por ejemplo, la expresión `entero1 + entero2` contiene el operador de suma + y los dos operandos `entero1` y `entero2`.

La división / entre dos números enteros devuelve un entero y el operador de módulo, que devuelve el resto de la

división entera, puede ser aplicado solamente a dos operandos enteros.

Podemos utilizar paréntesis `()`, de forma similar que lo hacemos en las expresiones algebraicas, para dar precedencia en las operaciones.

## OPERADORES Y EXPRESIONES RELACIONALES

Las condiciones juegan un papel fundamental en las estructuras de programación y son imprescindibles para poder efectuar programas y para la implementación de las estructuras de selección y de iteración.

Las condiciones nos permiten hacer *preguntas o evaluar* el *estado* de los *datos* empleados en un programa y de acuerdo al resultado de esa evaluación tomar decisiones sobre las acciones a desarrollarse. En efecto, en un programa y para resolver un problema, no todas las acciones deben ejecutarse, algunas solo se ejecutarán si ciertas condiciones están dadas, si se *satisfacen* una o más *condiciones*.

Es decir que para ejecutar algunas acciones podrán ponerse condiciones para ello. Estas condiciones pueden satisfacerse o no, lo que quiere decir que las respuestas posibles a las preguntas sobre el estado de los datos pueden ser solo dos: *si* o *no*, o también podría decirse *verdadero* (*true*, en inglés), o *falso* (*false*, en inglés), o también, ya que las computadoras se manejan con el sistema de numeración binario, 1 ó 0. Es por ello que, a las condiciones en computación, se les puede aplicar el Cálculo Proposicional (ver Apéndice A). Las condiciones son llamadas *expresiones condicionales*, o *expresiones lógicas*, o *expresiones booleanas*.

Una condición, una expresión lógica, estará compuesta de *operadores* y de *operandos*, que pueden ser variables, constantes u otras expresiones más complejas.

Los operadores más comunes, implementados en la mayoría de los lenguajes de programación, son los *operadores relacionales*, que como su nombre lo indica son los que nos permiten comparar dos operandos; por ejemplo el operador *igual*, que pregunta si dos operandos son iguales.

En la Tabla 3. 4 se muestran los operadores relacionales de C.

**Tabla 3. 4 Operadores relacionales en C**

| Operadores relacionales algebraicos | Operadores relacionales en C | Ejemplo de uso |
|---|---|---|
| = | == | `x == y` |
| ≠ | != | `x != y` |
| ≤ | <= | `x <= y` |
| ≥ | >= | `x >= y` |
| > | > | `x > y` |
| < | < | `x < y` |

Muchos lenguajes de programación cuentan con un tipo de dato particular para representar los valores *lógicos* o *booleanos* verdadero y falso, (o true y false; ó `1` y `0`). En C esto no ocurre; los valores de verdad son representados por dos valores enteros: el `0` para falso y el `1` para verdadero. Así, por ejemplo, las siguientes las expresiones relacionales siguientes devolverán los valores siguientes:

`1 == 1` → `1` (verdadero)

`10 <= 5` → `0` (falso)

`3 != 4` → `1` (verdadero)

Hay que notar dos cosas importantes en C:

Toda expresión que devuelva un valor entero distinto de `0` también será evaluada como verdadera. Por ejemplo, supongamos que una variable `i` tiene almacenado el número `10`, así al ejecutarse la sentencia

```
if (i) printf("verdadero\n");
```

imprimirá `verdadero` por pantalla.

Como las expresiones relacionales devuelven valores enteros (y no booleanos), estas pueden aparecer dentro de expresiones aritméticas. Por ejemplo, `40 + (3 != 4)` evaluará a `41`.

## OPERADORES Y EXPRESIONES LÓGICAS

Cada vez que queremos escribir condiciones compuestas, necesitamos hacer uso de los *operadores lógicos*, que son los que implementan las *conectivas lógicas* más comunes: la conjunción (∧), la disyunción (∨) y la negación (¬). Para esto, C cuenta con tres operadores lógicos, como se muestra en la Tabla 3. 5.

**Tabla 3. 5 Operadores lógicos**

| Operador lógico | Operador lógico en C | Ejemplo de uso | Significado |
|---|---|---|---|
| ∧ and | `&&` | `i != 0 && j > 1` | `i` distinto de `0` y `j` mayor que `1` |
| ∨ or | `\|\|` | `c == 'a' \|\| n == 0` | `c` igual al carácter `'a'` o `n` igual a `0` |
| ~ ¬ not | `!` | `! valido` | la variable `valido` no es verdadera (igual a `0`) |

Un aspecto importante a tener en cuenta es la forma en que se evalúan las expresiones lógicas en C. Estas son evaluadas de izquierda a derecha y la evaluación se detiene tan pronto como el valor de verdad pueda ser determinado, es decir, el segundo operando es evaluado sólo si es necesario. Por ejemplo,

`0 && …` → `0`

`1 || …` → `1`

## OPERADOR DE ASIGNACIÓN

Emplear un dato cuyo valor cambia implica usar una *variable* que, en computación, equivale a darle un nombre a un lugar en la memoria, donde se almacenará el valor que se necesite. Esto en contrapartida con el uso de *constantes* o *literales* que son justamente valores que no cambian.

La acción de *asignación* es la que permite *dar*, *asignar*, un valor a una *variable*.

La asignación requiere que haya un elemento capaz de almacenar la información o el dato asignado. Este elemento es una *variable*. En general, el valor que se asigna debe ser del mismo tipo que el elemento sintáctico que lo recibe. Sin embargo, hay asignaciones que se aceptan que sean de tipos distintos, aunque 'compatibles', en cuyo caso ocurren *conversiones implícitas* entre tipos. Otras veces, estas conversiones pueden ser forzadas de forma *explícita*, como en el caso del operador *cast* en C, que presentamos en la sección siguiente.

Las asignaciones son las que permiten que una misma variable almacene diferentes valores en distintos momentos durante la ejecución de un programa. En C, el operador de asignación es el signo igual (=).

Asumiendo las siguientes declaraciones de variables y constantes en C:

```
#define K -4
int i ;
```

algunos ejemplos de asignaciones válidas en C son los siguientes:

```
i = K * 2;
i = i + 5;
```

En la mayor parte de los lenguajes imperativos, la asignación es una sentencia. Sin embargo, en C es una operación, es decir, retorna un resultado, que en este caso es el valor asignado a la variable de la parte izquierda de la asignación. Esto permite hacer cosas tales como lo siguiente:

```
r = (i = K + 1) + 4;
```

Esto asignará `-3` a la variable `i`, este valor será retornado por la asignación interna al que se le sumará `4` devolviendo `1` que será finalmente asignado a la variable `r`.

Ya veremos que, si bien la asignación en C es una operación, al agregarle el punto y coma final, también es una sentencia.

Algunas asignaciones en C, tienen una versión compacta. Las más importantes, son mostradas en la Tabla 3. 6.

**Tabla 3. 6. Operadores de asignación aritméticos**

| Operador de asignación compacto | Ejemplo | Versión descomprimida | Resultado asignado a variable a (asumiendo la declaración `int a = 3;`) |
|---|---|---|---|
| += | a+=4 | a = a + 4 | 7 |
| -= | a-=4 | a = a - 4 | -1 |
| *= | a*=4 | a = a * 4 | 12 |
| /= | a/=4 | a = a / 4 | 0 |

## CONVERSIONES DE TIPOS – OPERADOR CAST

La *conversión de tipos* son distintas maneras de cambiar un tipo de dato en otro tipo de dato. Cada lenguaje tiene sus propias reglas de conversión de tipos.

Estas conversiones, en general, ocurren en las expresiones; en consecuencia, pueden ocurrir cuando se espera una expresión de otro tipo. Por ejemplo, en el cálculo de una expresión aritmética cuando un operador tiene operandos de tipos diferentes; en una asignación cuando el tipo de la expresión de la parte derecha es distinto del de la izquierda; en el pasaje de parámetros cuando el tipo del parámetro actual es diferente del formal.

En la mayor parte de los lenguajes de programación, se emplea la palabra *coerción* para denotar las *conversiones implícitas*, es decir, aquellas que son llevadas de manera automática ya sea que ocurran en tiempo de ejecución o de compilación. Por ejemplo, mezclar en una expresión aritmética valores enteros y reales, como ser 0.5 + 4, donde el valor entero es llevado, por el compilador, a real y luego sumado.

Las conversiones explícitas son aquellas donde el programador fuerza de forma explícita, por medio de alguna construcción sintáctica del lenguaje, el cambio de tipo. En algunos lenguajes, tales como en C, se emplea la palabra

*casting* para hablar de las *conversiones explícitas*.

## CONVERSIONES IMPLÍCITAS EN C

Las conversiones implícitas en C trabajan de una manera natural, como uno espera que sean: en general, el tipo 'menor' es *promovido* al tipo 'superior' antes de efectuarse la operación. El resultado es del tipo superior.

Por ejemplo,

```
int i = 5;
float f = 3.0;
float y;

f = i; /* el valor de i es promovido a float antes de asignarlo a f */
y = f + i; /* el valor de i es promovido a float antes de sumarlo con f */
```

Las reglas de conversiones implícitas del C se pueden consultar en el Apéndice A.6 del libro de Kerninghan & Ritchie, segunda edición. La Tabla 3. 7 muestra los tipos de datos en orden de jerarquía de mayor a menor junto con los especificadores de conversión de `printf` y `scanf`.

**Tabla 3. 7. Especificadores de conversión para `printf` y `scanf`.**

| Tipo de Dato | Especificador de conversión `printf` | Especificador de conversión `scanf` |
|---|---|---|
| long double | %Lf | %Lf |
| double | %f | %lf |
| float | %f | %f |
| unsigned long int | %lu | %lu |
| long int | %ld | %ld |
| unsigned int | %u | %u |
| int | %d | %d |
| unsigned short int | %hu | %hu |
| short int | %hd | %hd |
| char | %c | %c |

## CONVERSIONES EXPLÍCITAS EN C: OPERADOR CAST

En C, las conversiones explícitas de tipos pueden ser forzadas en cualquier expresión por medio del operador unario *cast*, cuya sintaxis es

(*nombre-de-tipo*) *expresión*

donde *expresión* es convertido al tipo *nombre-de-tipo* de acuerdo a las reglas dadas en el Apéndice A.6 del libro de Kerninghan & Ritchie, segunda edición.

El efecto preciso del operador cast es equivalente al efecto de una conversión implícita como si *expresión* fuese asignada a una variable cuyo tipo es *nombre-de-tipo*. Por ejemplo, asumiendo las siguientes declaraciones

```
float resultado;
int i = 9;
int j = 5;
```

las asignaciones siguientes producirán distintos resultados:

```
resultado = i / j;  /* resultado queda con 1.0 */
resultado = (float)i / j;  /* resultado queda con 1.8 */
resultado = (float)(i / j);  /* resultado queda con 1.0 */
resultado = (float)i / (float) j;  /* resultado queda con 1.8 */
```

En la primera, se calcula la división entera ya que los dos operandos (`i` y `j`) son enteros y cuyo resultado es `1`; luego el valor `1` es promovido por medio de una conversión implícita a tipo `float` que es el tipo de la variable resultado, almacenándose `1.0`.

En la segunda asignación, se crea una copia temporal en punto flotante del valor de `i`. El valor de `i` sigue siendo entero. A continuación se promueve el operando `j` a `float` y se efectúa la división en punto flotante. Este resultado (`1.8`) es almacenado en la variable `resultado`.

En la tercera, se realiza la división entera que resulta en `1` y se lleva el resultado a `float` aplicándole el cast `(float)`. Ese resultado (`1.0`) es almacenado en la variable `resultado`.

En la cuarta asignación, se llevan ambos operandos a `float` por medio de los dos operadores cast y luego se realiza la división en punto flotante y se almacena en la variable resultado.

### OPERADORES DE INCREMENTO Y DECREMENTO

Existen dos operadores, propios del lenguaje C, usados para incrementar y decrementar en `1` las variables. El operador `++` suma `1` a su operando, mientras que el operador `--` le resta `1`. Ambos operadores tienen versiones *prefijo* y *posfijo*, es decir, pueden ir antes o después del operando. La Tabla 3. 8 muestra todas las posibilidades así como el significado de cada una de ellas.

**Tabla 3. 8. Operadores de incremento y decremento**

| Operador | Explicación | Ejemplo de uso | Efecto |
|---|---|---|---|
| `a++` | Devuelve el valor de `a` y luego lo incrementa en `1`. | `i = a++;` | Asigna el valor de `a` a la variable `i` y luego incrementa a en `1`. Es equivalente a<br>`i = a;`<br>`a = a + 1;` |
| `++a` | Incrementa el valor de `a` en `1` y devuelve ese valor. | `i = ++a;` | Incrementa en `1` la variable `a` y ese valor lo asigna a la variable `i`. Es equivalente a<br>`a = a + 1;`<br>`i = a;` |
| `a--` | Devuelve el valor de `a` y luego lo decrementa en `1`. | `printf("%d", a--);` | Imprime el valor de `a` y luego decrementa en `1` su valor. Es equivalente a<br>`printf("%d", a);`<br>`a = a - 1;` |
| `--a` | Decrementa el valor de `a` en `1` y devuelve ese valor. | `printf("%d", --a);` | Decrementa en `1` la variable `a` y luego muestra su valor por pantalla. Es equivalente a<br>`a = a - 1;`<br>`printf("%d", a);` |

## 3.5 ESTRUCTURAS DE CONTROL

A lo largo del tiempo, los lenguajes que implementan el paradigma de *programación imperativa*, han ido incorporando no solo los principios de dicha programación sino también elementos metodológicos que ayudan a no cometer errores, aún cuando no a evitarlos totalmente. No todos los lenguajes tienen incorporados algunos de estos elementos metodológicos que pertenecen a lo que se ha dado en llamar *programación estructurada*. Así, la programación estructurada podría ser vista como un subconjunto de la programación imperativa.

Muchos de los principios de la programación estructurada coinciden con principios generales de resolución de problemas. Otros son consecuencia de la necesidad de evitar errores que la estructura sintáctica de algunos lenguajes puede llevar a cometer al programador. Así, por ejemplo, está el principio que dice que los datos deben definirse antes de empezar a utilizarlos (hay lenguajes que obligan, sintácticamente, a hacerlo); o el principio que el programa debe ejecutarse en el mismo orden en que se lea, evitando saltos que alteren el flujo normal de ejecución y de lectura del programa.

Este último principio responde al hecho que muchos lenguajes de programación tienen incorporadas órdenes o sentencias de *bifurfación*, que permiten que la ejecución del programa se traslade, "salte", haga una bifurcación a otro sector del mismo y, en consecuencia, la ejecución del programa no siga el orden de lectura del mismo, haciendo difícil su lectura y puesta a punto.

El uso y abuso de las sentencias de bifurcación en los programas llevó a lo que se suele referir como "código spaghetti". A raíz de este problema, C. Böhm y G. Jacopini, en 1966, escribieron un trabajo en donde se demostraba que los programas podían ser escritos sin el uso de este tipo de sentencia. En 1968, Edsger W. Dijkstra publicó también un trabajo en donde se criticaba el uso excesivo de la sentencia de bifurcación *goto* y abogaba por lo que se dio en llamar la programación estructurada. En ella se propone el uso de tres *estructuras de control* (la *secuencia*, la *selección* y la *iteración*) para escribir cualquier programa, pudiéndose así reducir al mínimo o incluso eliminar el uso de las sentencias de bifurcación.

Los lenguajes de programación imperativos tienen las estructuras sintácticas necesarias para implementar todas las estructuras de control. Asimismo, todos los lenguajes imperativos cuentan con una sentencia u operación de asignación, que es una acción básica de la programación imperativa, la cual permite cambiar los valores de ciertas posiciones de la memoria de la computadora denotados por un identificador de variable.

Cabe notar que los lenguajes imperativos, como por ejemplo Pascal, C, FORTRAN, COBOL, lenguajes assembly (de ensamblado), etc. permiten el uso de otras sentencias para entrada/salida, de invocación de módulos, entre otras.

Las estructuras sintácticas de programación a las que hacemos referencia en esta sección tienen que ver con las estructuras que controlan el flujo de ejecución de un programa. Estas acciones se estructuran en los lenguajes de programación por medio de elementos sintácticos llamados *frases*, *sentencias* o *enunciados*, cada una de las cuales es una orden a ejecutar. Es decir que encontraremos, en la mayor parte de los lenguajes imperativos, una estructura sintáctica para construir sentencias, frases o enunciados para la selección, para la iteración y para la secuencia.

La mayor parte de los lenguajes de programación permiten no sólo las *sentencias simples*, donde cada sentencia equivale a una orden, sino también *sentencias compuestas*, que son sentencias que incluyen una o más sentencias simples. A esto se lo conoce también como *bloque de sentencias* o simplemente *bloque*.

## SECUENCIA

La estructura de control de *secuencia* aprovecha el orden de lectura tradicional de los idiomas occidentales –es decir de izquierda a derecha y de arriba hacia abajo– para implementarse.

La ejecución de las sentencias se hace en un orden dado: en forma *secuencial*, es decir una después de la otra, y no se ejecuta la segunda sentencia hasta que la primera haya terminado de ejecutarse, y así sucesivamente. Esta secuencia sigue el orden normal de lectura de izquierda a derecha y de arriba a abajo.

En consecuencia, la estructura de secuencia de la programación estructurada se efectúa respetando la normal ejecución secuencial que todos los lenguajes de programación imperativa tienen, sin emplear sentencias de salto. Es decir que el orden de lectura de un programa es idéntico con el orden de ejecución del mismo. O por lo menos, eso es lo que se espera si no cuenta con sentencias de bifurcación.

En C, se usa el punto y coma como un *finalizador de sentencia* y no un *separador de sentencia* como lo es en otros lenguajes (por ejemplo, en Pascal). Así, expresiones, como por ejemplo,

```
y = 4
 j++
scanf(…)
```

se vuelven sentencias cuando se las sigue por un punto y coma,

```
y = 4;
j++;
scanf(…);
```

Las llaves (`{` y `}`) se usan para agrupar declaraciones y sentencias y formar una *sentencia compuesta* o *bloque*. Por ejemplo, las llaves que encierran las sentencias del cuerpo de la función `main` (y de cualquier otra función) son un ejemplo. También pueden aparecer en las sentencias `if`, `while`, `for` y otras.

Las variables en C, pueden ser declaradas dentro de *cualquier* bloque. No va punto y coma detrás de la llave que cierra un bloque, aunque muchos compiladores lo aceptan.

## SELECCIÓN

La estructura de *selección* evalúa una *expresión condicional* (expresión lógica) para decidir si una sentencia (simple o compuesta) se ejecutará o no. Si la expresión condicional al ser evaluada, resulta verdadera la sentencia (simple o compuesta) se ejecutará, sino, si la expresión condicional al ser evaluada, resulta falsa la sentencia no se ejecutará. En ambos casos, ya sea que se ejecuten o no, la o las sentencias controladas por la selección, se continúa ejecutando la secuencia establecida: es decir, se ejecuta la sentencia que sigue en el orden de ejecución a la sentencia de selección.

La mayor parte de los lenguajes de programación implementan la sentencia de selección permitiendo que si la expresión condicional resulta falsa se ejecute otra u otras sentencias para esa alternativa. Es decir que si la expresión condicional es verdadera se ejecutan una o más sentencias, sino, si es falsa, se ejecutan otra u otras sentencias. En cualquiera de los dos casos, una vez finalizada la ejecución de la selección, se continúa ejecutando la sentencia que sigue en el orden secuencial a la sentencia de selección.

En general, podemos decir que una sentencia de selección en C tiene la siguiente forma:

`if (`*condición*`)` *sentencia*-1`;`
`else` *sentencia*-2`;`

Donde *sentencia*-1 puede ser una sentencia simple o compuesta y es la que se ejecuta si *condición* evalúa a *verdadero*, es decir toma un valor distinto de `0`. A su vez, *sentencia*-2 puede ser una sentencia simple o compuesta y es la que se ejecuta si *condición* evalúa a *falso* (`0`). *sentencia*-1 y *sentencia*-2 se conocen como '*ramas de la selección*': la rama del `if` o rama del verdadero y la rama del `else` o rama del falso.

Muchos lenguajes de programación permiten que la rama del falso sea opcional, es decir que puede estar presente o no. Estas selecciones suelen ser llamadas selecciones de rama vacía. C no es una excepción y su formato general es:

`if (`*condición*`)` *sentencia*-1`;`

Es de hacer notar que una sentencia de selección se ejecuta siempre, al menos para evaluar su condición. Dependiendo del resultado de la evaluación de la condición se ejecutará siempre una de las dos ramas, si tiene dos

ramas. Si tiene sólo una rama, de acuerdo con el resultado de la evaluación de la condición, podrá ejecutarse o no la rama del verdadero.

Los siguientes son ejemplos de sentencias de selección válidas en C:

```
if (a==1 && b) a++;
```

En esta selección de rama vacía, si la variable `a` vale `1` y `b` es verdadero (es decir tiene un valor distinto de `0`) entonces se ejecutará el incremento en `a`, caso contrario se seguirá con la ejecución de la sentencia siguiente al `if`.

```
if (a == 1 && b) {
    a++;
    b = 0;
}
```

En este caso, sólo se ejecutará la sentencia compuesta (las sentencias entre llaves `{}`) en la rama del verdadero si la expresión condicional `a == 1 && b` evalúa a verdadero, en caso contrario se seguirá con la sentencia siguiente al `if`.

```
if (a == 1 && b) {
    a++;
    b = 0;
}
else a =0;
```

Este ejemplo muestra una selección con dos ramas. Si la expresión `a == 1 && b` es verdadera entonces se ejecutará la sentencia compuesta sino la sentencia de la rama del falso, que en este ejemplo es una sentencia simple (una asignación) aunque también, de ser necesario, podría haber sido compuesta.

Las sentencias de selección pueden *anidarse*. Esto significa que tanto *sentencia*-1 como *sentencia*-2 pueden a su vez contener sentencias de selección. Un ejemplo posible sería el de la Figura 3.4 donde *sentencia*-1 corresponde a una sentencia condicional de dos ramas.

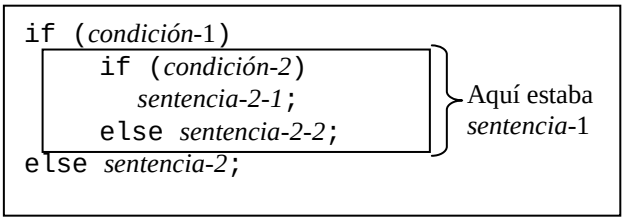


**Figura 3.4 Ejemplo de sentencias de selección anidadas**

Dado que la parte del `else` es opcional, puede existir ambigüedad en una sentencia `if` anidada cuando se omite uno de los `else`. Esto es resuelto en C, asociando el `else` al `if` más interno. Por ejemplo,

```
if (n > 0)
    if (a >b)
        x = a;
    else
        x = b;
```

se asociará el `else` con el `if (a > b)`. Si ese no es el efecto deseado, se deberá indicar usando llaves:

```
if (n > 0){
    if (a >b)
        x = a;
}
```

```
else
    x = b;
```

Existe una generalización de la sentencia de selección, con la que muchos lenguajes de programación cuentan, que es una *sentencia de selección múltiple* en donde, dependiendo del valor de una expresión, se ejecuta una de las alternativas de un grupo de alternativas dadas.

C permite el uso de la sentencia `switch` para selecciones múltiples. Esta consiste de una serie de etiquetas de caso (`case`) y de un caso opcional `default`. La forma general es:

```
switch (expresión) {
    case constante1: sentencias;
    case constante2: sentencias;
    …
    case constanten: sentencias;
    default: sentencias;
}
```

*constante*$_1$… *constante*$_n$ deben ser constantes enteras. En primer lugar se evalúa la *expresión* del `switch`. Si este valor es igual a algunas de las constantes de los casos, entonces se ejecutan las sentencias asociadas al caso. Los casos se comparan de arriba hacia abajo. Si ninguno de las constantes coincide con el valor de *expresión* entonces se ejecutan las sentencias asociada al caso `default`.

Dado que el caso `default` es opcional, si este no se encontrase y ninguna constante coincidiera con el valor de la expresión, ninguna de las sentencias del `switch` se ejecutaría.

Una vez que se ejecutó un caso, se siguen ejecutando los casos que se encuentran a continuación a menos que se haya colocado una sentencia `break.` Por ejemplo, el siguiente trozo de programa:

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

suponiendo que la variable `i` vale `1`, el programa imprimirá:

```
uno
dos
tres
```

Para que imprima solo el valor de la variable `i` en letras, deberá colocarse una sentencia `break` que fuerza a salir de la sentencia `switch`:

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

Las alternativas `case` pueden aparecer vacías, pudiéndose simular múltiples valores para una alternativa como, por ejemplo, en el siguiente trozo de programa:

```
switch (ch) {
    case ‘,’:
    case ‘;’:
```

```
    case '.': printf("signo de puntuacion\n");
              break;
    default: printf("otro caracter que no es un signo de puntuacion\n");
}
```

## ITERACIÓN

Iterar es repetir. La acción o sentencia de iteración es la que permite *repetir* la ejecución de una o más *sentencias*. Es la que permite comandar la repetición de una o más acciones tantas veces como sea necesario. La iteración se conoce también bajo otros nombres: *ciclo*, *bucle* (españolización del francés *boucle* que significa aro o rulo), o en inglés *loop*.

En una repetición, cada vez que se ejecuta la o las sentencias comandadas por la iteración, se dice que se *pasa* por el ciclo (o bucle, o loop), o se *hace* un ciclo (o bucle, o loop), o se *ejecuta* un ciclo (o bucle, o loop).

Una iteración, en general, tiene dos partes: la *condición* que controla la repetición y el *bloque de la iteración*.

Decimos que la *condición* es la parte de la iteración que controla la iteración porque es la que nos dice cuántas veces (o hasta cuándo) se hará la repetición.

Llamamos *cuerpo* o *bloque de iteración* a la o las sentencias cuya ejecución es controlada por la condición de la iteración. Así, diremos que es el cuerpo de la iteración el que se ejecuta repetidamente.

Generalmente, toda iteración, antes de ejecutarse, necesita de algún tipo de *inicialización*. Esto significa que a algunas variables, que forman parte tanto de la condición como del bloque de iteración, deberá asignárseles un valor inicial que es con el que se comienza a ejecutar la iteración en sí misma. Por lo general, esto debe hacerse en forma explícita justo antes de la iteración, sin embargo, algunas veces, las variables que forman parte de la condición de la iteración ya "vienen" de antes con los valores correctos y necesarios.

La condición que comanda la iteración es una expresión lógica, que sirve para *controlar* las veces que se va a ejecutar el cuerpo de la iteración. En efecto, cuando se desea repetir una o más acciones un número determinado de veces y no indefinidamente, debe darse la condición para que se *desarrolle* la repetición, o la condición para que se *detenga* la repetición. En el primer caso diremos que la iteración ejecutará el bloque de iteración *mientras* la *condición* sea *verdadera* (o falsa, depende de cómo se encuentre implementada la iteración en el lenguaje de programación); y en el segundo caso, diremos que la iteración ejecutará el bloque de iteración *hasta* que la *condición* sea *verdadera* (o falsa, depende de cómo se encuentre implementada la iteración en el lenguaje de programación).

Si la ejecución del bloque de la iteración no se realiza, la iteración termina y se continúa con la ejecución secuencial, es decir que se ejecuta la sentencia siguiente, en el orden secuencial, a la sentencia de iteración.

Si la ejecución del bloque de la iteración se realiza, al fin de dicha ejecución, se volverá a evaluar la condición, y así sucesivamente hasta que la condición evalúe al valor de verdad que controla el fin de la iteración.

En general, los lenguajes de programación proveen dos variantes de iteración dependiendo del momento en que se evalúe la condición que controla la iteración. La evaluación de la condición puede hacerse *antes* de ejecutar el bloque de iteración o *después* de ejecutarlo. Esto es significativo para la primera vez que se ejecuta la iteración. En efecto, si la primera vez que se ejecuta la iteración se debe *primero* evaluar la condición podría resultar que siendo esta falsa (por ejemplo, si ese es el valor de verdad de finalización) no se ejecute nunca el bloque de iteración. Pero si por el contrario, se evalúa la condición *después* de ejecutar el bloque de iteración, aún cuando la condición sea falsa se habrá ejecutado, al menos una vez, el bloque de iteración.

Esto tiene importancia para determinar cuántas veces puede llegar a ejecutarse el bloque de la iteración. En efecto, si se evalúa la condición *antes* de ejecutar el bloque de la iteración, diremos que el bloque puede ejecutarse

*cero o más veces* (algunas veces se dice, aunque no sea del todo correcto, que la iteración puede ejecutarse cero o más veces), esto es así ya que al evaluar la condición por primera vez podría ser que esta tome el valor que implica el fin de la iteración y en consecuencia el bloque de la iteración no se ejecute nunca.

Por el contrario, si se evalúa la condición después de ejecutar el bloque de la iteración, diremos que el bloque de la iteración puede ejecutarse *una o más veces* (al igual que antes, aunque no sea del todo correcto, algunas veces se dice que la iteración puede ejecutarse una o más veces). En efecto, al evaluar la condición después de la primera ejecución del bloque podría ser que esta evaluara al valor que implica el fin de la iteración, sin embargo el bloque de la iteración ya se ejecutó una vez.

Como ya dijimos, la ejecución de la iteración es controlada por una condición (expresión lógica); esta es la encargada de determinar el fin de la misma, es decir cuándo terminará la repetición de la ejecución del bloque de la iteración. Una vez evaluada la expresión lógica, si esta toma un valor que implica la continuación de la repetición, sólo podrá cambiar su valor de verdad si dentro del bloque de la iteración hay al menos una variable, que forma parte de su condición, que cambia su valor, haciendo que, a su vez, cambie el valor de verdad de la condición. A partir de esto, podemos deducir algunos principios elementales, que si bien no son suficientes, son necesarios para que la repetición termine:

*Debe haber al menos una variable en la expresión lógica que controla la iteración*. Por ejemplo, en una iteración que se ejecuta mientras su condición es verdadera, si la expresión lógica que controla la iteración es, por ejemplo, 1=1, esta expresión lógica (uno es igual a uno) será siempre verdadera. La repetición no se detendrá jamás (a menos que se desenchufe la computadora o alguna otra operación similar).

*Al menos una de las variables que forman parte de la expresión lógica que controla la iteración, debe ser modificada dentro del bloque de la iteración*. Es decir que dentro del bloque de la iteración debe haber una o más sentencias que modifiquen una o más variables que forman parte de la condición para que el valor de verdad que tome la condición pueda variar.

Por otro lado existen otras consideraciones que son importantes para el funcionamiento correcto, o mejor dicho para que la iteración haga lo que se pretende de ella:

Es conveniente establecer claramente no sólo cuál es la condición que controla la iteración, sino también la negación de dicha condición, que es la condición de parada de la iteración y cómo quedan las variables de la condición cuando la iteración ha terminado. Esto es importante ya que muchas veces, al terminar la iteración podemos encontrarnos con valores inesperados en las variables que forman parte de la condición de la iteración, ya sea porque han sido cambiados inadvertidamente o porque asumíamos que tendrían otros valores al terminar la iteración.

Es importante, también, conocer los valores en que se encuentran todas las variables, de la condición y del bloque, antes de iniciarse la iteración y una vez terminada la misma, y muchas veces también es importante conocer los valores de las variables cada vez que se ejecuta el bloque de iteración.

C cuenta con tres variantes de sentencias de repetición: la sentencia `while`, la sentencia `do/while` y la sentencia `for`. Sin embargo, la sentencia `while` es la más general ya que con ella se puede simular el efecto de las otras dos.

La sentencia de repetición `while` tiene la siguiente forma:

`while` (*expresión*)
    *sentencia*;

donde *sentencia* puede ser una sentencia simple o un bloque de sentencias. Aquí, *sentencia* se repetirá mientras la condición dada en *expresión* sea verdadera. Es decir, al evaluar *expresión*, si es distinta de `0` (es verdadera) se ejecutará *sentencia* y se reevaluará la *expresión*. Esto se repite hasta que la evaluación de *expresión* sea `0`; en ese

punto se sigue con la ejecución de la sentencia que sigue al `while`. Por ejemplo, el programa de la Figura 3.5 calcula la suma de los primeros *n* números naturales, donde *n* es un valor ingresado por el usuario, incluido el `0`.

```
#include <stdio.h>

main()
{
    int suma, n, i;

    /* Pide que se ingrese n mientras el valor de n sea menor a 0 */
    n = -1;
    while (n < 0) {
          printf("Ingrese un numero natural n mayor o igual a cero: ");
          scanf("%d", &n);
   }

    suma = 0;
    i = 0;
    while (i <= n){
          suma = suma + i;
          i = i + 1;
   }
    printf("La suma de los %d primeros naturales es %d\n", n, suma);
}
```

**Figura 3.5. Versión 1 del programa que calcula la suma de los n primeros naturales.**

La sentencia de repetición `for` en C tiene la siguiente forma:

`for` (*expresión1*; *expresión2*; *expresión3*)
    *sentencia*;

donde *sentencia* puede ser una sentencia simple o una compuesta (bloque de sentencias). Las expresiones *expresión1* y *expresión3* son asignaciones o llamados a funciones y *expresión2* es una condición.

En primer lugar se ejecuta *expresión1*, luego se evalúa *expresión2*; si es verdadera entonces se ejecutan *sentencia* y luego *expresión3* en ese orden. Este proceso continúa hasta que *expresión2* se vuelve falsa.

La sentencia `for` es equivalente a

*expresión1*;
`while` (*expresión2*) {
    *sentencia*;
    *expresión3*;
}

El uso de `for` o `while` es una cuestión de gusto. Sin embargo, el `for` se prefiere cuando existe una inicialización simple de una variable y un incremento en el cuerpo. Por ejemplo, el cálculo de la suma podría también ser programado usando la sentencia `for` como se ve en el programa de la Figura 3.6.

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

```
    int suma, n, i;

    /* Pide que se ingrese n mientras el valor de n sea menor a 0 */
    n = -1;
    while (n < 0) {
          printf("Ingrese un numero natural mayor o igual a cero: ");
          scanf("%d", &n);
    }

    suma = 0;
    for (i = 0; i <= n; i++)
          suma = suma + i;

    printf("La suma de los %d primeros numeros naturales es %d\n", n, suma);
}
```

**Figura 3.6. Versión 2 del programa que calcula la suma de los n primeros naturales.**

Como vimos antes, las sentencias de repetición pueden evaluar la condición al principio de la misma o al final después de realizar un paso por el cuerpo de la misma, asegurándose la ejecución de las sentencias del cuerpo al menos una vez. La sentencia de iteración `do/while`, cuya forma general es la siguiente, corresponde a este tipo de sentencia:

```
do
    sentencia;
while (expresión)
```

En primer lugar se ejecuta *sentencia* y a continuación se evalúa la condición *expresión*. Si *expresión* es verdadera se vuelve a ejecutar *sentencia* y así sucesivamente. Cuando la condición se hace falsa se termina la iteración y se continúa con la ejecución de la sentencia siguiente al `do/while`.

El programa de la Figura 3.7 muestra una nueva versión del programa de la suma de los primeros naturales. En él se ha cambiado el `while` de la primera iteración que controla que el usuario ingrese un número mayor o igual a cero por un `do/while`, cuyo uso en ese contexto resulta más natural.

```
#include <stdio.h>
main()
{
    int suma, n, i;

    do {
          printf("Ingrese un numero natural mayor o igual a cero: ");
          scanf("%d", &n);
    } while (n < 0); /* repite mientras n sea menor a 0 */

    suma = 0;
    for (i = 0; i <= n; i++)
          suma = suma + i;

    printf("La suma de los %d primeros naturales es %d\n", n, suma);

}
```

**Figura 3.7. Versión 3 del programa que calcula la suma de los n primeros naturales.**

# 4 MODULARIDAD

Un principio ampliamente difundido e importante para resolver problemas es el de "*dividir para reinar*", es decir separar el problema en partes, en problemas más pequeños, para así enfrentarlos y resolverlos en un tamaño más accesible y fácil de manejar.

Este principio tiene una directa aplicación en la programación de computadoras, para lo cual muchos lenguajes de programación cuentan con una característica adecuada: la *modularidad*. Estos lenguajes permiten escribir programas compuestos de porciones o partes que se conocen como *módulos*, muchas veces nombrados también como *subprogramas*, *subrutinas*, *procedimientos*, *funciones*, entre otros. En todos los casos, los principios generales y elementales que los gobiernan son los mismos, aún cuando algunos tengan características distintas y mayores o menores prestaciones: todos permiten separar un programa en partes más pequeñas y reusar esos trozos de programa o módulos ya escritos para resolver problemas similares.

Un módulo de programa tiene dos partes bien diferenciadas: la *definición* del módulo y la *invocación o llamada* al módulo.

La *definición* de un módulo es donde se dice qué es lo que el mismo hará, es decir, cuáles serán sus efectos al ser invocado, es decir cuando se ejecute.

Una *invocación o llamada* ocurre cuando se usa el módulo. En el punto del programa donde se hace la invocación al módulo es donde, en tiempo de ejecución del programa, se ejecutarán las acciones comprendidas en la definición del módulo.

En un programa, por cada módulo, habrá una única definición, ya que bastará con una vez para decirse qué es lo que el módulo hace. Sin embargo, puede haber más de una invocación al mismo, ya que una solución puede usarse tantas veces como sea necesario. Por supuesto, podría no usarse nunca, es decir no invocarse nunca, pero entonces, ¿para qué se definió?

## 4.1 LAS FUNCIONES EN C

En C, los módulos se llaman *funciones*. Un programa C, típicamente, se crea combinando el uso de *funciones definidas por el usuario*, que son funciones escritas por el programador, y *funciones predefinidas*, que son las que ya vienen en las bibliotecas del lenguaje C. Si bien, estas últimas, no forman parte estrictamente del lenguaje, siempre vienen provistas con el compilador, brindando al programador un gran número de funciones, de entrada/salida, para manejo de strings, de caracteres, para cálculos matemáticos, entre otras. Por ejemplo, las funciones `printf` y `scanf`, que hemos usado en los ejemplos anteriores, son funciones predefinidas de la biblioteca estándar `stdio.h`.

Las funciones definidas por el usuario, que son las que trataremos en este capítulo, se definen en el programa una única vez y se *invocan* por medio de una *llamada* a la función tantas veces como sea necesario.

La ejecución de un programa se hace siempre de forma secuencial, comenzando por la función `main`. Cada vez que se invoca una función, el flujo de ejecución abandona la función que se está ejecutando en ese momento y salta a la función invocada. En cuanto la función invocada termina, el flujo de ejecución retoma el punto en que se produjo la llamada y continúa la ejecución por donde se quedó.

Supongamos, para ejemplificar, que queremos un programa que imprima las primeras `10` potencias de `2` y de `3`. Para ello, definiremos una función entera `potencia` que devuelva, como resultado, el cálculo de un entero `n` elevado a una potencia entera positiva `m`, e invocaremos la función `potencia` desde `main` con los correspondientes argumentos.

```
#include <stdio.h>

int potencia(int, int);  /* prototipo */

main()
{
    int i ;

    for (i = 0; i <= 10 ; i++)
          printf("2 a la %d: %d - 3 a la %d: %d \n", i, potencia(2, i),
                i, potencia(3, i));
    return 0;
}

/* Función que eleva la base n a la potencia m (m >= 0)*/
int potencia(int n, int m)
{
    int i, p;

    p = 1;
    for (i = 0; i < m; i++)
          p = p * n;
    return p;
}
```

La función `main` invoca dos veces a la función `potencia` en la línea

```
printf("%d %d %d \n", i, potencia(2, i), potencia(3, i));
```

En cada llamada se pasan dos *parámetros actuales* o *reales* (o *argumentos*) a `potencia` y esta retorna un valor entero, el que es impreso en pantalla.

La primera línea de la definición de la función `potencia`,

```
int potencia(int n, int m)
```

declara el tipo y nombre de cada uno de sus *parámetros formales* así como el tipo del resultado que devuelve la función.

La función `potencia` retorna un valor entero (`int`) a la función invocante (en este caso a `main`). Esto ocurre al ejecutarse la sentencia

```
return p;
```

la cual retorna el valor de la variable `p` y restituye el control a la función `main`.

Sin embargo, no todas las funciones necesitan retornar un valor. En este caso, como veremos en la sección siguiente, el tipo de retorno de la función debe declararse de tipo `void`.

Hay dos maneras de restituir el control a la función invocante sin devolver valor alguno. Una forma de hacerlo es por medio de la sentencia

```
return;
```

y la otra es omitiendo la sentencia `return`, al alcanzarse en la ejecución la llave derecha de terminación del bloque o cuerpo de la función.

La segunda línea del programa,

```
int potencia(int, int);  /* prototipo */
```

corresponde a la *declaración del prototipo* de la función `potencia`. En el prototipo de una función se deben declarar los tipos de los parámetros así como el tipo que retorna la función. Esta declaración indica al compilador el tipo de dato devuelto por la función, el número de parámetros que espera recibir, el tipo de los parámetros y el orden en el cual estos se esperan. El prototipo es usado por el compilador para controlar cada una de las invocaciones a la función. Debe, por supuesto, también coincidir con la definición de la función. Los nombres de los parámetros en el prototipo no necesitan coincidir, de hecho pueden omitirse, como en el ejemplo.

## DEFINICIÓN DE UNA FUNCIÓN EN C

La forma general de la definición de una función es:

*tipo-de-retorno identificador*(*lista-opcional-de-parametros-formales*)
{
    *declaraciones*
    *sentencias*
}

donde *identificador* es el nombre que le damos a la función y que usaremos en la invocación a la misma, pudiendo ser cualquier identificador válido.

El *tipo-de-retorno* es el tipo del valor que retorna la función. El tipo `void` es usado como *tipo-de-retorno* para indicar que la función no retorna ningún valor. Si se omite el *tipo-de-retorno*, se asume que es `int` (esto ocurre, por ejemplo, con la función `main` del programa anterior).

Si se observa, al final de `main`, hay una sentencia `return 0;`. Puesto que `main` es una función que devuelve un `int` también puede retornar un valor a quien la invoca que, por tratarse de `main`, será siempre el ambiente en donde el programa se ejecuta.

La *lista-opcional-de-parametros-formales* es una lista separada por comas con los tipos y los nombres de sus *parámetros formales*, la cual puede estar vacía, es decir, una función podría no tener parámetros, en cuyo caso esto se indica colocando la palabra `void` en lugar de la lista de parámetros.

A continuación, entre llaves, viene el *cuerpo* de la función o *bloque* en donde se pondrán las *declaraciones* y *sentencias* necesarias para que la función haga su cometido.

## INVOCACIÓN DE UNA FUNCIÓN

La invocación de una función se hace empleando el nombre de la función junto con los *parámetros actuales*, si los tuviera. En caso de no tener parámetros, simplemente se colocará el nombre seguido de `()`.

Una función puede ser invocada en un lugar del programa en donde vaya una expresión del mismo tipo retornado por la función o, incluso, la función invocante a otra función puede ignorar el valor retornado y hacer la invocación en un lugar donde va una sentencia. Cuando el tipo de retorno de una función es `void`, es decir, no retorna ningún valor, una invocación a ella siempre deberá ocurrir en un lugar donde va una sentencia.

## PASAJE DE PARÁMETROS

Los módulos de un programa (o funciones, en C) se comunican entre sí por medio de los datos que son pasados como parámetros en la invocación y los datos que son devueltos como resultado de las invocaciones.

Un módulo, generalmente, realizará alguna tarea empleando datos que le son provistos. Hay distintas maneras de

proveerle los datos que el módulo necesita durante su ejecución. Una de las formas más usuales es a través del uso de *parámetros*.

Estos parámetros son declarados en la definición del módulo, en donde se dice cuántos y de qué tipo son los datos que el módulo espera cuando el mismo sea invocado, así como con qué nombres se puede referenciar esos datos dentro del módulo. Estamos hablando de variables que tomarán un valor cuando el módulo sea invocado. Estas variables se conocen como *parámetros formales*.

Al invocarse un módulo, habrá que darle o *pasarle* los valores reales con los que este trabajará: uno por cada parámetro declarado en la definición del mismo. Estos valores los tomarán los parámetros formales. En la invocación del módulo estos valores se conocen como *parámetros reales* o *actuales*. Se habla así de *pasarle*, al módulo, los parámetros reales, es decir los datos concretos que el módulo deberá procesar.

Algunos lenguajes llaman *argumentos* a los parámetros. Otros reservan la palabra *argumento* para los parámetros formales y parámetros para los parámetros reales. Nosotros, para evitar confusiones hablaremos de *parámetros formales* y de *parámetros reales* o *actuales*.

La forma en que se realiza la asociación entre los parámetros formales y los reales se la conoce como *pasaje de parámetros*. Los parámetros pueden ser *pasados* al módulo de distintas maneras. Dos de las más comunes o usuales son el pasaje que se llama por *valor* o *constante* y el que se llama por *dirección* o *referencia* o *variable*.

En el pasaje por valor, lo que se pasa al módulo es una copia del dato. Es decir, el valor del parámetro real es copiado en el parámetro formal, de ahí que se denomine por valor. Todo cambio que se haga al parámetro formal dentro del módulo no se verá reflejado en el parámetro real.

En el pasaje por dirección o referencia o variable sería equivalente a pasar la variable propiamente dicha donde se encuentra el dato y no una copia del dato. Lo que se pasa, en realidad, es la dirección del parámetro real. En consecuencia, todo cambio que se haga sobre el parámetro formal se verá reflejado en el parámetro real. Es por esta razón que cuando la declaración de un parámetro formal es hecha por dirección, el correspondiente parámetro actual no puede ser una constante, debe ser siempre una variable.

Una imagen que puede evocarse para distinguir ambos métodos de pasaje de parámetros es la que equivale a decir que, si uno tiene un cuaderno con notas tiene dos formas de "pasarle" esas notas a un compañero: le da una fotocopia de las notas (éste es el pasaje por valor) o le da el cuaderno mismo (pasaje por dirección). Pero claro, en éste último caso, si el compañero no cuida el cuaderno puede ser que éste termine escrito con otras cosas, manchado o inclusive se pierda.

En C, todas las llamadas a funciones son por valor. Sin embargo, es posible simular el pasaje por dirección, para lo cual la función que hace el llamado deberá pasar la dirección de la variable que será modificada por la función invocada, mientras que la función invocada deberá definir el correspondiente parámetro formal como de tipo puntero. Como los punteros serán estudiados más adelante, por ahora nos limitaremos a dar una "receta" de cómo simular el pasaje por dirección en C. Para ilustrar las dos variantes de pasaje de parámetros, analicemos primero el programa de la Figura 4.1, el cual intercambia el valor de dos variables `a` y `b`:

```
#include <stdio.h>

void intercambia(float, float);

main()
{
    float a = 1.0;
    float b = 2.0;

    printf("a = %f, b = %f\n", a, b);
    intercambia(a, b);
```

```
    printf("a = %f, b = %f\n", a, b);
}

void intercambia(float x, float y)
{
    float temp;
    temp = x;
    x = y;
    y = temp;
}
```

**Figura 4.1. Pasaje por valor.**

Si ejecutamos el programa, la salida del mismo será:

```
a = 1.000000, b = 2.000000

a = 1.000000, b = 2.000000
```

Es decir, no se logra el efecto deseado de intercambiar los valores de las variables `a` y `b` a pesar que dentro de la función los valores fueron intercambiados. Esto es porque, en realidad, los que se han intercambiado son los valores de las copias de `a` y `b` (es decir, `x` e `y`) pero no los valores de `a` y `b`. Para lograr el efecto deseado, los parámetros deberían pasarse por dirección o referencia, como se muestra en el programa de la Figura 4.2. En el capítulo 5, luego de introducir el tipo puntero, explicaremos cómo funciona en C el pasaje de parámetros por dirección.

```
#include <stdio.h>

void intercambia(float *x, float *y);

main()
{
    float a = 1.0;
    float b = 2.0;

    printf("a = %f, b = %f\n", a, b);
    intercambia(&a, &b);
    printf("a = %f, b = %f\n", a, b);
}

void intercambia(float *x, float *y)
{
    float temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

**Figura 4.2. Pasaje por dirección.**

## 4.2 ALCANCE O ÁMBITO DE LOS IDENTIFICADORES

El *ámbito* o *alcance de un identificador* es el trozo de un programa en el cual el identificador es válido y puede ser usado, es decir, en *donde* su declaración tiene efecto. Por ejemplo, en C, el alcance de una variable declarada dentro de un bloque de sentencias es el mismo bloque, fuera de él la variable no puede ser referenciada.

Cada lenguaje tiene sus propias reglas de ámbito. En el caso del lenguaje C, podemos distinguir cuatro tipos de

alcances diferentes para un identificador: *alcance de archivo*, *alcance de bloque*, *alcance de prototipo* y *alcance de función*.

Todo identificador declarado por fuera de cualquier función tiene *alcance de archivo*. Esto significa que es accesible (o conocido) en el archivo desde el punto en donde ha sido declarado hasta el final del archivo. Estos identificadores suelen ser llamados *globales*. Las variables globales, las definiciones de funciones y los prototipos de funciones tienen este alcance. Por ejemplo, en el programa de la Figura 4.2, `intercambia` y `main` son identificadores de funciones y pueden ser referenciados desde el punto en donde aparecen en el archivo y hasta el final del mismo.

Los identificadores con *alcance de bloque* son aquellos que han sido declarados dentro de un bloque o dentro de la lista de parámetros formales en una definición de función. Son accesibles sólo desde el punto de su declaración o definición y hasta el final del bloque que los contiene. Tales identificadores suelen ser llamados *locales.* Por ejemplo, en el programa de la Figura 4.2, `a` y `b` son variables locales a `main` y solo son visibles desde su declaración hasta el final de `main`. Lo mismo ocurre con `x`, `y` y `temp`, que son locales a `intercambia` y en consecuencia son visibles sólo dentro del bloque o cuerpo de la función `intercambia`.

En C, cualquier bloque puede contener declaraciones de variables locales. Por lo tanto, cuando haya bloques anidados y un identificador declarado en un bloque externo tuviese el mismo nombre de un identificador declarado en uno de los bloques internos, el identificador externo será ocultado por el interno hasta el final del bloque interno.

Los únicos identificadores que tienen *alcance de prototipo* son los identificadores usados como nombres de parámetros en una declaración de un prototipo. Dado que esos identificadores no se usan para nada más que a efectos de documentación para el programador e incluso pueden obviarse, su alcance es sólo la declaración del prototipo, en consecuencia pueden ser usados en cualquier otro punto del programa sin generar ambigüedades. Por ejemplo, las variables `x` y `y` de la declaración del prototipo de la función `intercambia` no tienen nada que ver con las variables `x` y `y` de la declaración de la función. Incluso, podría habérseles llamado `a` y `b` o dado otros nombres.

Finalmente, los únicos identificadores en C que tienen *alcance de función* son las *etiquetas*. Las etiquetas son declaradas implícitamente cuando se las usa en una sentencia y tienen como alcance toda la función en donde aparecen. Se usan en conjunción con las sentencias `goto`. Sin embargo, dado que es posible programar sin el uso de sentencias `goto`, no usaremos etiquetas ni `goto`.

## 4.3 TIEMPO DE VIDA DE LOS IDENTIFICADORES

El *tiempo de vida o permanencia* de un identificador es el período durante el cual el identificador existe en la memoria de la computadora y no necesariamente coincide con su alcance.

En C, los identificadores globales, como las variables globales y los nombres de funciones, tienen una *permanencia estática* en la memoria, es decir, existen desde el momento en que se inicia la ejecución del programa y hasta que este termina (el compilador les asigna memoria antes que se comience con la ejecución de la función `main` y no la libera hasta que `main` termina). Sin embargo, eso no significa que puedan ser accedidos en cualquier punto del mismo (recodemos que el ámbito no es lo mismo que el tiempo de vida): sólo pueden ser accedidos a partir del punto en que han sido declarados.

En C, los identificadores locales o internos, tienen una *permanencia automática.* En consecuencia, sólo las variables locales de una función (las declaradas en la lista de parámetros o las declaradas dentro de un bloque) tienen permanencia automática. Esto significa que son creadas cada vez que se entra en el bloque en el que se encuentran declaradas, existen mientras el mismo está activo y se destruyen cuando se abandona el mismo al encontrar la llave (}) o al alcanzar una sentencia `return`.

Se puede forzar que la permanencia de una variable local sea estática anteponiéndole en su declaración el especificador `static`. De esta manera, la variable local conservará su valor al salir del bloque en el cual ha sido declarada. Así, en una invocación sucesiva de la función, la variable contendrá el valor con el cual se encontraba al finalizar la invocación anterior.

El programa de la Figura 4.3, cuya salida es mostrada en la Figura 4.4, ilustra los distintos ámbitos y tiempos de vida.

```
#include <stdio.h>

void a(void);     /* prototipo de función */
void b(void);     /* prototipo de función */
void c(void);     /* prototipo de función */

int x = 1;        /* variable global */

int main()
{
    int x = 5;  /* variable local a main */
    printf("Local x en el ambito externo de main = %d\n", x);
    {    /* comienza un nuevo ámbito de bloque en main*/
          int x = 7;
          printf("Local x en el ambito interno de main = %d\n", x);
    }    /* fin del nuevo ámbito de bloque*/
    printf("Local x en el ambito externo de main = %d\n", x);
    a();          /* a() tiene una variable local automática */
    b();          /* b() tiene una variable local estática */
    c();          /* c() usa la variable global */
    a();          /* a() reinicializa su variable local */
    b();          /* b() x local estática conserva el valor de la */
                  /*invocación anterior */
    c();          /* la variable global también conserva su valor */
                  /* precedente */
    printf("Local x en el ambito externo de main = %d\n", x);
    return 0;
}


void a(void)
{
    int x = 25;   /* inicializada cada vez que a() es llamada */
    printf("Local x en a() despues de entrar a a() = %d\n", x);
    x++;
    printf("Local x en a() antes de salir de a() = %d\n", x);
}


void b(void)
{
    static int x = 50;   /* la inicialización ocurre solo la primera */
                         /* vez que b() es llamada */
    printf("Local x en b() despues de entrar a b() = %d\n", x);
    x++;
    printf("Local x en b() antes de salir de b() = %d\n", x);
}
```

```
void c(void)
{
    printf("Global x despues de entrar a c() = %d\n", x);
    x*=10;
    printf("Global x antes de salir de c() = %d\n", x);
}
```

**Figura 4.3. Ejemplo de ámbitos**

```
Local x en el ambito externo de main = 5
Local x en el ambito interno de main = 7
Local x en el ambito externo de main = 5
Local x en a() despues de entrar a a() = 25
Local x en a() antes de salir de a() = 26
Local x en b() despues de entrar a b() = 50
Local x en b() antes de salir de b() = 51
Global x despues de entrar a c() = 1
Global x antes de salir de c() = 10
Local x en a() despues de entrar a a() = 25
Local x en a() antes de salir de a() = 26
Local x en b() despues de entrar a b() = 51
Local x en b() antes de salir de b() = 52
Global x despues de entrar a c() = 10
Global x antes de salir de c() = 100
Local x en el ambito externo de main = 5
```

**Figura 4.4. Salida del programa de la Figura 4.3.**

# 5 PUNTEROS

Un *puntero* (o *pointer*, en inglés) en la mayor parte de los lenguajes de programación, y en el C en particular, es exactamente lo que su nombre indica: un apuntador, es decir un tipo provisto por el lenguaje para "apuntar" a algo. En realidad, lo "apuntado" es un lugar en la memoria (una dirección) de la computadora donde puede almacenarse un valor de un tipo determinado. Así, por ejemplo, hablaremos de un apuntador a enteros cuando la variable puntero apunta a una posición donde se almacena un entero, o de una variable de tipo puntero a real que podrá almacenar la dirección donde se guardará un valor real.

Una variable de tipo puntero no contiene otra cosa que la dirección (también podemos decir la *referencia*) de un lugar en la memoria. Normalmente, una variable hace *referencia directa* a un valor específico, mientras que una variable puntero lo hace de forma *indirecta*, es decir, hay una *indirección* (Figura 5. 1).

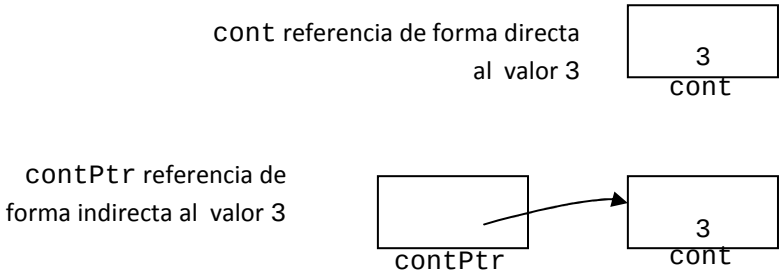


**Figura 5. 1. Referencia directa (variable `cont`) y referencia indirecta (variable `contPtr`)**

Así, podemos decir que la variable `contPtr` apunta a la dirección de memoria de la variable `cont`, o simplemente que apunta a `cont`, y que la dirección donde se almacenan los valores que tomará `cont` es apuntada por el puntero `contPtr`, o simplemente que `cont` es apuntada por `contPtr`.

Los tipos de datos punteros son considerados tipos *dinámicos* por oposición a los otros tipos de datos que se consideran *estáticos*. Esto se refiere al hecho de que en la declaración de una variable estática, por ejemplo una variable de tipo entero, el compilador reservará la cantidad de bits necesarios para almacenar un valor del tipo en cuestión, en este caso un entero. Como ya hemos visto, este lugar existirá en la memoria durante todo el tiempo de vida de la variable. Por el contrario, como veremos en la sección 5.3, una variable de tipo puntero, por ejemplo puntero a carácter, podrá "crearse" y "destruirse" tantas veces como sea necesario, durante la ejecución del programa, es decir, *dinámicamente*, podrá asignársele y desasignársele memoria.

## 5.1 PUNTEROS EN C

En C, la declaración

```
int i = 10;
```

hará que el compilador reserve un lugar en la memoria, por ejemplo la dirección `5600`, en donde se almacenará el valor corriente de la variable `i`. La Figura 5.2 muestra una posible representación de la variable entera `i`.

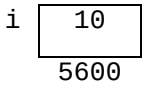


**Figura 5.2. Variable `i`**

Una variable de tipo puntero tomará valores que son referencias (direcciones o apuntadores) a posibles valores de un tipo dado o un valor particular llamado puntero nulo, representado en C por la constante `NULL`, definida en la

biblioteca estándar `stdio.h`, la cual indica que la variable puntero no apunta a nada. Así, por ejemplo la declaración en C de una variable `ptr` de tipo puntero a entero que apunta inicialmente a `NULL` tomará la siguiente forma:

```
int *ptr = NULL;
```

Esto hará que el compilador reserve un lugar en memoria para `ptr` el cual contendrá el valor `NULL`. La Figura 5.3 muestra dos posibles representaciones del puntero `ptr`.


**Figura 5.3. Dos representaciones distintas de la variable `ptr` de tipo puntero a entero**

En C, los punteros también pueden inicializarse con `0`, que es equivalente a inicializarlo con `NULL`. Otra posibilidad sería inicializarlo con una dirección; en este caso, dado que se trata de un puntero a entero, podríamos inicializarlo con la dirección de una variable entera, por ejemplo, con la dirección de la variable `i`. C provee el *operador de dirección* &, que permite obtener la dirección de una variable. Por ejemplo, la declaración

```
int *ptr = &i;
```

declara el puntero `ptr` a entero y lo inicializa con la dirección de la variable entera `i`, es decir, `ptr` apunta a la dirección de memoria `5600` de `i` o simplemente decimos que `ptr` apunta a `i`. La Figura 5.4 muestra dos representaciones posibles de esta situación.

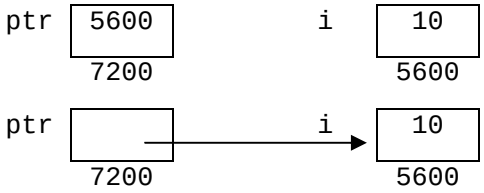


**Figura 5.4. El puntero `ptr` apunta a `i`**

El operador `*` de *desreferenciación o indirección* permite acceder al valor apuntado por una variable de tipo puntero. Por ejemplo, el siguiente trozo de código mostrará el valor `10` apuntado por `ptr`, lo modificará y luego volverá a imprimir el nuevo valor, `8`, apuntado por `ptr`.

```
printf("valor apuntado por ptr %d\n", *ptr);
*ptr = 8;
printf("valor apuntado por ptr %d\n", *ptr);
```

Notemos que si imprimimos ahora la variable `i`, su valor habrá cambiado a `8`.

Las variables de tipo puntero pueden ser asignadas siempre que el tipo de la parte derecha de la asignación sea el mismo tipo del de la parte izquierda, como se ve en el programa de la Figura 5.5 en donde, además, se puede ver el uso de punteros y de los operadores & y *. La salida del programa se muestra en la Figura 5.6.

```
#include <stdio.h>

int main()
{
    int *p, *q, i;

    i = 7;
    p = &i;
    q = p;
```

```
    *p = *p + 1;
    printf("%d %d %d\n", *p, *q, i);
    i++;
    printf("%d %d %d\n", *p, *q, i);
    return 0;
}
```

**Figura 5.5. Ejemplo de uso de punteros**

```
8 8 8
9 9 9
```

**Figura 5.6. Salida del programa de la Figura 5.5.**

## PUNTEROS A `void`

Un puntero a `void` es un *puntero genérico* que puede representar cualquier tipo de puntero, y que puede apuntar a objetos de datos de cualquier tipo. Cualquier puntero puede ser asignado a un puntero a `void` y un puntero a `void` puede ser asignado a cualquier puntero.

Son útiles cuando uno necesita un puntero que apunte a datos de diferentes tipos en diferentes momentos. Por ejemplo:

```
int i;
char c;
void *dato;

i = 6;
c = 'a';
dato = &i;   /* dato apunta a la dirección donde comienza i */
dato = &c;   /* dato apunta a la dirección donde comienza c */
```

Dado que el compilador no sabe cuál es el tamaño del dato apuntado por un puntero a `void`, no podemos desreferenciarlo si no es haciéndolo con el uso de un operador cast. Así, por ejemplo, si quisieramos mostrar a qué apunta el puntero genérico `dato`, debemos decirle al compilador, por medio de un cast, cuál es el tipo apuntado por el puntero genérico, como por ejemplo:

```
int i;
char c;
void *dato;

i = 6;
c = 'a';
dato = &i;
printf("dato apunta al valor entero %d\n", *(int*) dato));
dato = &c;
printf("dato apunta al valor de tipo caracter %c\n", *(char*) dato);
```

Hemos visto, en la sección anterior, que es posible asignar punteros entre sí cuando apuntan a datos de un mismo tipo. Usando punteros a `void` podemos hacer asignaciones con punteros a otros tipos, por ejemplo como se ve en la

```
#include<stdio.h>

int main(){
        char i = 0;
```

```
        char j = 0;
        char* p = &i;
        void* q = p;
        int* pp = q;   /* inseguro pero legal en C, no en C++ */

        printf("%d %d\n",i,j);
        *pp = -1;      /* sobrescribe la memoria comenzando en &i */
        printf("%d %d\n",i,j);
}
```

Figura 5.7.

```
#include<stdio.h>

int main(){
        char i = 0;
        char j = 0;
        char* p = &i;
        void* q = p;
        int* pp = q;   /* inseguro pero legal en C, no en C++ */

        printf("%d %d\n",i,j);
        *pp = -1;      /* sobrescribe la memoria comenzando en &i */
        printf("%d %d\n",i,j);
}
```

**Figura 5.7. Uso de punteros genéricos.**

Si bien el programa de la figura es sintácticamente correcto, no hace un manejo seguro de los tipos de datos, como se ve en la sentencia de asignación

```
int* pp = q;
```

en donde `pp`, que es un puntero a entero, queda apuntando a la dirección a donde apunta `q`, que en este momento se encuentra apuntando a la dirección de `i`, que es un `char`. Los efectos de usar un puntero a un tipo que no apunta a algo de ese tipo pueden ser desastrosos. Al ser `q` un puntero genérico (`void *`) no sabemos a cuántos bloques de memoria se encuentra apuntando. Por lo tanto, hay que ser muy cuidadoso con el uso que se hace de los punteros genéricos.

## 5.2 PASAJE DE PARÁMETROS POR REFERENCIA O DIRECCIÓN EN C

Como ya habíamos adelantado, el pasaje de parámetros en C es por valor. Sin embargo, es posible simular el pasaje por dirección o referencia haciendo uso de punteros.

La forma de eliminar esta restricción es *pasar por valor la dirección* de la variable que se desea modificar, es decir, pasar un apuntador a dicha variable. En otras palabras, lo que pasamos es una copia de la dirección de la variable y no una copia de la variable, y es a través de esta dirección que se puede modificar la variable y así simulamos el pasaje por dirección.

Dado que las variables de tipo puntero almacenan direcciones, para pasar la dirección de una variable de un tipo *T*, lo que hacemos es declarar el parámetro como un puntero a *T* y cuando invocamos a la función, usamos el operador & para obtener la dirección de la variable que se desea modificar. A su vez, dentro de la función, para acceder al contenido del parámetro usamos el operador `*`de desreferenciación.

En el caso de los arreglos, como veremos en el capítulo 7, no deberemos usar el operador & en la invocación, ni

tampoco el operador * dentro del cuerpo de la función. Esto es porque en C el nombre del arreglo es equivalente a la dirección de la posición base del mismo y más aun, como veremos, existe una equivalencia entre los arreglos y los punteros.

En el ejemplo siguiente resolveremos un mismo problema usando pasaje por valor y por dirección. Se trata de un programa que dada una letra la transforma a mayúscula. En la primera versión del programa, mostrado en la Figura 5.8, la función `main` pasa por valor a la función `toUpperByValue` la variable `letra` de tipo `char` con la letra a transformar y devuelve como resultado, por medio de la instrucción `return`, la correspondiente letra en mayúscula. Este valor será asignado en `main` a la variable `letra`.

En la segunda versión del programa (Figura 5.9), la función `main` pasa la dirección de la variable `letra` a la función `toUpperByReference`. Esta función recibirá como parámetro un puntero a `char` llamado `letraPtr`. La función resolverá la referencia y asignará el nuevo valor en la dirección apuntada por `letraPtr` modificando al mismo tiempo el valor de la variable `letra` en `main`.

```
#include <stdio.h>

char toUpperByValue(char);

int main()
{
    char letra = 'a';

    printf("El valor inicial de la variable letra es %c\n", letra);
    letra = toUpperByValue(letra);
    printf("El nuevo valor de la variable letra es %c\n", letra);
    return 0;
}

char toUpperByValue(char c)
{
     if (c >= 'a' && c < 'z') /* es una letra minuscula */
           c = c - 32;        /* la cambia a mayúscula */
     return c;
}
```

**Figura 5.8. Transforma a mayúscula una letra usando pasaje por valor.**

```
#include <stdio.h>

void toUpperByReference(char *);

int main()
{
    char letra = 'a';

    printf("El valor inicial de la variable letra es %c\n", letra);
    toUpperByReference(&letra);
    printf("El nuevo valor de la variable letra es %c\n", letra);
    return 0;
}

void toUpperByReference(char *letraPtr)
{
   if (*letraPtr >= 'a' && *letraPtr < 'z') /* si es una letra minuscula*/
```

```
            *letraPtr = *letraPtr - 32;      /* la cambia a mayúscula */
}
```

**Figura 5.9. Pasa a mayúscula una letra usando pasaje por referencia.**

La Figura 5.10 muestra la salida de los dos programas, que en este caso es la misma, mientras que la Figura 5.11 y la Figura 5.12 muestran el efecto de la ejecución de los programas sobre las variables.

```
El valor inicial de la variable letra es a
El nuevo valor de la variable letra es A
```

**Figura 5.10. Salida de los programas de las Figura 5.9 y Figura 5.8.**

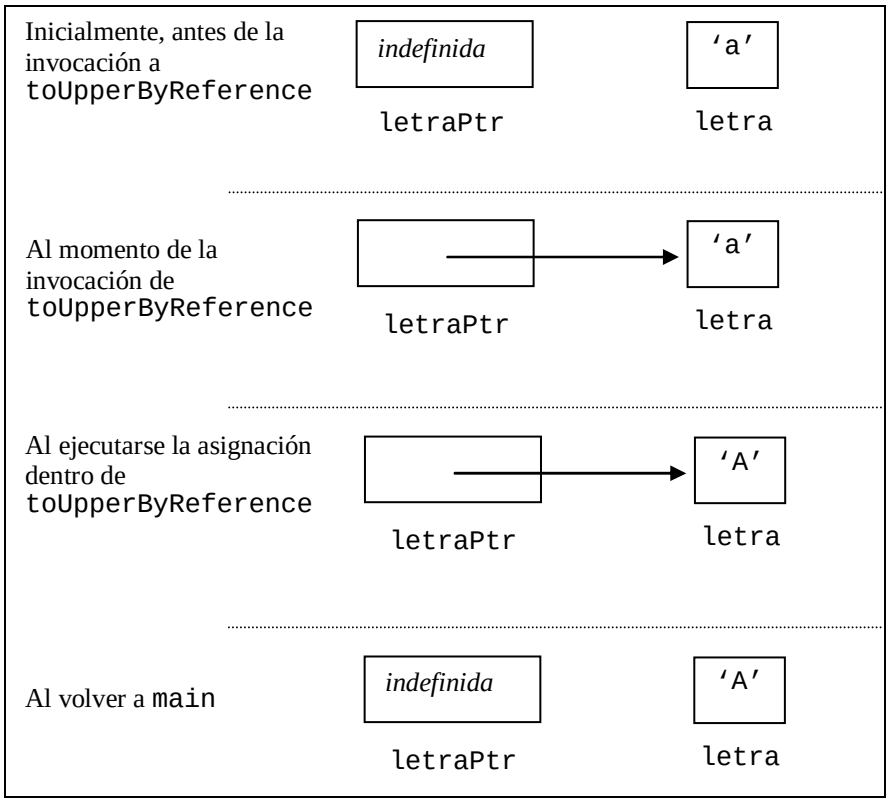


**Figura 5.11. Efecto sobre las variables de la ejecución del programa de la Figura 5.9.**

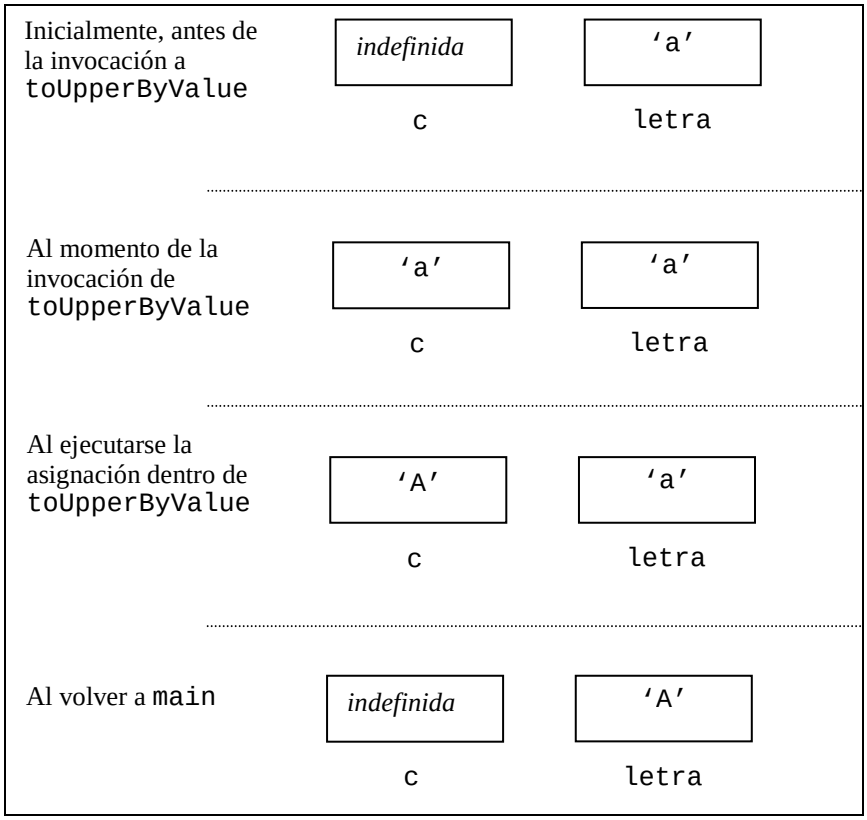


**Figura 5.12. Efecto sobre las variables de la ejecución del programa de la Figura 5.8.**

En general, una buena práctica de programación es usar las funciones pasando los datos por valor, para evitar efectos colaterales no deseados. Sin embargo, recordemos que en el caso de los arreglos en C, por economía de espacio, estos son siempre pasados por referencia.

## 5.3 ASIGNACIÓN DINÁMICA DE LA MEMORIA

Como ya vimos en la sección 4.3, las variables tienen un tiempo de vida, es decir, el tiempo durante el cual se garantiza que la variable existe y tiene memoria asignada.

Aparte del tiempo de vida o duración estática, que tienen tanto las variables globales como las variables locales declaradas con el especificador `static`, y de la duración automática, propia de las variables locales, existen variables que tienen una *duración asignada*.

La *duración asignada* se refiere a las variables cuya memoria se reserva de forma dinámica, es decir, memoria que se crea y se debe liberar de forma explícita en tiempo de ejecución, haciendo que el tiempo de vida de las variables dependa del momento en que se les asigna y desasigna memoria.

La biblioteca estándar de C proporciona las funciones `malloc`, `calloc`, `realloc` y `free` para el manejo de memoria dinámica con punteros. Estas funciones están definidas en el archivo de cabecera `stdlib.h` y nos permiten que el programa pida memoria y la libere de acuerdo a sus necesidades. El límite de la memoria asignada puede ser tan grande como la cantidad de memoria física disponible en la computadora para la aplicación en cuestión.

Las funciones `malloc`, `free` y el operador unario `sizeof` son fundamentales para el manejo dinámico de la memoria.

`malloc` toma como argumento el número de bytes que se solicitan y retorna un puntero a `void` (`void *`) que apunta a la memoria asignada. Un puntero a `void` puede ser asignado a una variable de cualquier tipo puntero usando el operador cast. Si no hay suficiente memoria para asignar, entonces `malloc` retorna `NULL`, lo que permite controlar en el programa si se pudo realizar o no la asignación de memoria.

Normalmente, `malloc` es usada en combinación con `sizeof` para indicar la cantidad de bytes a pedir. `sizeof` es un operador que recibe como operando un tipo de dato y retorna el tamaño que ocupa en memoria un objeto del tipo de dato al cual se le aplicó el operador. Por ejemplo, `sizeof(float)` me permitirá conocer cuántos bytes ocupa un `float`. En el contexto de la asignación dinámica de memoria es extremadamente útil ya que nos permitirá solicitar la cantidad exacta de memoria. Por ejemplo,

```
float *newPtr = (float *) malloc(sizeof(float));
```

evalúa `sizeof(float)` para determinar el tamaño en bytes de un dato de tipo `float`, asigna una nueva área de memoria con esa cantidad de bytes y devuelve un puntero a esa área el cual es almacenado en la variable `newPtr`. Si no hubiese habido suficiente memoria para asignar, `malloc` retornaría `NULL`.

Por otro lado, la función `free` desasigna memoria. Esto significa que la memoria apuntada por el puntero al que se le aplica `free` es retornada al sistema y en consecuencia la misma podrá ser reasignada en el futuro. Por ejemplo, si quisiéramos liberar la memoria previamente solicitada por la sentencia `malloc`, haremos

```
free(newPtr);
```

Esto nos será de suma utilidad a la hora de implementar estructuras de datos con datos recursivos, manejadas con asignación dinámica de la memoria, como veremos más adelante.

# 6 ESTRUCTURAS DE DATOS

## 6.1 INTRODUCCIÓN

Emplear un dato, cuyo valor cambia, implica usar una *variable*, que en programación imperativa equivale a darle un nombre a un lugar de la memoria, donde se almacenará el valor que se necesite. Esto en contrapartida con el uso de *constantes* que son justamente valores que no cambian.

Hasta ahora hemos hablado de almacenar en una variable un único valor simple, de un tipo dado, a la vez. Sin embargo, más de una vez será necesario conservar más de un valor. Esto es fácil de realizar empleando variables para cada uno de los valores que sean necesarios.

Sin embargo, hay veces que estos datos, por alguna razón, tienen una unidad conceptual y conviene mantenerlos juntos y no separados en variables distintas. Por ejemplo, todos los datos que corresponden a una persona, tales como su edad, el estado civil, su nombre y apellido, dirección, documento de identidad, etc. Así, estos datos, deberían, entonces, poder ser agrupados en una unidad, que llamaremos *estructura de datos*.

Una *estructura de datos* es un conjunto de datos que se agrupan por alguna razón. Este agrupamiento lleva implícito un almacenamiento de cada uno de los componentes de la estructura.

Llamaremos *elementos* de la estructura a cada uno de sus componentes, es decir, a cada uno de los datos que forman parte de ella.

Una estructura de datos presupone un conjunto de elementos y tiene, al menos potencialmente, la posibilidad de tener más de un elemento, de lo contrario no podría hablarse de estructura.

Así entonces, cuando se tiene una estructura de datos, surgen distintos interrogantes sobre la estructura y su contenido (sus elementos):

¿Cómo *se incorpora* un nuevo elemento a la estructura?

¿Cómo *se elimina o cambia* un elemento que ya está en la estructura?

¿Cómo *se recupera* un elemento, ya sea para conocerlo o inspeccionarlo, para eliminarlo de la estructura o para buscarle un lugar a uno nuevo?

¿Cómo *se organizan* los elementos en la estructura? Es decir, ¿en qué *orden* se encuentran unos con respecto a los otros?

¿*Cuántos* elementos se pueden guardar? Esto tiene que ver con la *capacidad de almacenamiento* de la estructura.

¿Cómo se identifica o *selecciona* a los distintos elementos de la estructura? Es necesario poder distinguir, sin ambigüedades, cada uno de los elementos de la estructura. Llamamos a esto el *selector* de la estructura.

¿Qué *tipo de datos* pueden guardarse en la estructura? Esto es el tipo de datos de los elementos de la estructura. A este tipo de dato le llamaremos *tipo de dato base* de la estructura.

Las respuestas a estos interrogantes determinarán el tipo de estructura de datos de la cual se trate. Las tres primeras preguntas tienen que ver con las *operaciones* que pueden hacerse sobre la estructura de datos. Todas estas características no siempre se encuentran presentes en los lenguajes de programación. Algunos lenguajes tienen algunas de ellas, otros otras. Tampoco todos los lenguajes de programación tienen la capacidad de administrar todas las estructuras de datos que presentamos en los capítulos 8, 9 y 10. En estas notas presentamos una abstracción, si bien no de todas, de las principales estructuras de datos y sus características.

## 6.2 CAPACIDAD DE LAS ESTRUCTURAS DE DATOS

Una estructura de datos tiene una *capacidad* para almacenar elementos. La misma puede ser teóricamente infinita aunque en la práctica esto no será así.

Cuando la *capacidad* es *finita* la misma puede ser *dinámica* o *estática*. Así, hablaremos de estructuras de datos dinámicas y estructuras de datos estáticas dependiendo del tipo de capacidad que las mismas tengan.

### CAPACIDAD DINÁMICA

Hablamos de capacidad dinámica cuando la capacidad de la estructura, para almacenar elementos, *crece* o *disminuye* de acuerdo con las incorporaciones o eliminaciones, respectivamente, de elementos. Si bien la estructura crece con las inserciones y disminuye con las supresiones, tiene un límite máximo a su crecimiento, lo que está dado por la finitud de la capacidad de la estructura.

### CAPACIDAD ESTÁTICA

La capacidad estática implica que la capacidad de la estructura es constante ya sea que se incorporen o saquen elementos de la misma. Es decir que la estructura tiene un número fijo y determinado de elementos y la cantidad de los mismos *no varía* con las inserciones y supresiones.

## 6.3 OPERACIONES

Distinguiremos tres operaciones fundamentales que pueden realizarse sobre una estructura de datos y sus elementos:

COLOCAR un elemento nuevo en la estructura.

Para el caso de las estructuras *estáticas*, como esto implica que la misma no puede crecer, hablaremos entonces de *asignar* un valor a un elemento de la estructura, es decir, cambiar un valor por otro.

Para el caso de las estructuras *dinámicas* esto implica que la cantidad de elementos de la misma aumenta. Hablaremos, entonces, de una *inserción*.

SACAR, de la estructura, un elemento ya existente en ella.

Para el caso de las estructuras *estáticas* esto implica que la misma no disminuirá su tamaño. Hablaremos, en este caso, de una *asignación* que se hará sobre un valor ya existente, haciendo que éste último cambie su valor.

Para el caso de las estructuras *dinámicas* esto implica que la cantidad de elementos de la misma disminuye. Hablaremos, entonces, de una *supresión*.

INSPECCIONAR un elemento de la estructura, para conocer su valor.

Si bien la inspección se hace de a un elemento por vez, es decir, no puede hacerse simultáneamente con más de un elemento, sí puede hacerse, inspeccionando de a uno por vez, de forma sistemática sobre algunos o todos los elementos de la estructura.

La inspección de los elementos NO modifica el valor contenido en los mismos ni el tamaño de las estructuras. Se dice que la inspección (o lectura del valor contenido en un elemento) no es destructiva.

En realidad, para las estructuras estáticas no existe una diferencia notable entre las operaciones de COLOCAR y SACAR, simplemente se trata de *cambiar* un valor de un elemento por otro valor, es decir, de llevar a cabo una asignación de un nuevo valor.

En el caso de las estructuras dinámicas, como éstas crecen con las inserciones y disminuyen con las supresiones, pueden estar, en algún momento, *vacías* (por ejemplo si se han suprimido todos sus elementos o cuando no se ha insertado ninguno). Para estos casos será útil conocer si la estructura tiene al menos un elemento o está vacía. Esto se realiza por medio de un control o consulta sobre la estructura para saber si está vacía o no, o lo que es lo mismo si tiene al menos un elemento o no.

Esta consulta se trata en realidad de un *predicado lógico*, que aplicado sobre la estructura dinámica, devuelve verdadero o falso, a la pregunta de si la estructura está vacía.

Las estructuras de datos dinámicas, si bien desde un punto de vista teórico tienen una capacidad infinita, este no es el caso cuando las llevamos a la memoria de la computadora. Así, será también necesario contar con un predicado para saber si la estructura acepta que se le agreguen nuevos elementos, es decir, si no está *llena*.

La condición de estar vacía o no estar vacía es propia de las estructuras dinámicas. La condición de estar llena no es propia de la estructura dinámica sino de la forma en que estas son implementadas en la computadora.

## 6.4 ORDEN DE LOS ELEMENTOS

Cualquiera sea la estructura de datos, ésta tendrá un orden, entendiendo por orden alguna relación de orden entre los elementos de la estructura. Es decir, sus elementos estarán almacenados en la estructura en un orden dado, aún cuando no parezca que es así.

Muchas veces el orden de los elementos de una estructura es propio de la estructura en sí misma, otras veces, por encima del orden que la misma estructura impone, existe un orden que surge del *manejo* o *manipulaciones* que de la estructura y sus elementos se haga. Por ejemplo, en una cola frente a una ventanilla, el orden es, normalmente y si no hay colados, *el primero en llegar a la cola es el primero que es atendido*. Sin embargo, por encima de éste orden, la cola podría haber sido ordenada *alfabéticamente* de acuerdo con los apellidos de las personas en la misma. Todavía sería atendido el primero de la cola, sin embargo, ésta (la cola) habría sido manipulada o reconstituida, hecha de nuevo, para poder ordenarla alfabéticamente. De la cola original, se habría hecho una nueva cola poniendo primero en ella al que le correspondiese alfabéticamente estar primero. En esta nueva cola, ordenada alfabéticamente, habría ingresado primero, y sería atendido primero, el que le corresponda estar primero alfabéticamente.

En general, diremos que el orden de los elementos de las estructuras de datos puede ser *cronológico* y *no cronológico*.

### ORDEN CRONOLÓGICO

Cuando hablamos de *orden cronológico* es que queremos hacer referencia de alguna manera al tiempo. En efecto, en éste caso, el tiempo juega un papel en la forma que los elementos de las estructuras de datos se ordenan unos con otros. Este orden puede reconocerse que es impuesto, en algunas estructuras de datos, por la estructura misma.

El orden cronológico, a su vez, puede ser:

*Orden de llegada a la estructura*, es decir que los elementos se encuentran ordenados de acuerdo al tiempo, al momento en el tiempo, en el que ingresaron a la estructura.

*Orden de salida de la estructura*, es decir que los elementos se encuentran ordenados de acuerdo al tiempo, al momento en el tiempo, en que se encuentran listos para salir de la estructura.

*Orden de recorrida* o *inspección de la estructura*, es decir que los elementos se encuentran ordenados de acuerdo al momento en que son encontrados cuando se recorre o inspecciona la estructura.

### ORDEN NO CRONOLÓGICO

En este orden el tiempo no juega un papel significativo. Los elementos se encuentran ordenados sin ninguna referencia a alguno de los tipos de orden mencionados en la sección anterior, es decir que no se encuentran ordenados con relación al tiempo.

Pueden encontrarse ordenados por algún otro criterio. Por ejemplo, ordenados alfabéticamente o siguiendo el orden de los naturales, etc. Este orden puede reconocerse que no es impuesto en las estructuras de datos por la estructura misma, sino que es impuesto por una relación de orden externa entre sus elementos y que conlleva una manipulación de la estructura y de sus elementos.

## 6.5 SELECTOR DE LOS ELEMENTOS

El *selector* de una estructura sirve para seleccionar, elegir, identificar, a cada uno de los elementos de la estructura para poder realizar alguna operación sobre ellos: cambiarlo, inspeccionarlo, eliminarlo, etc.

El selector debe ser *unívoco*, es decir no puede permitirse que dentro de una estructura exista la posibilidad de ambigüedad en la elección de un elemento. Cada elemento debe poder ser *unívocamente* identificado.

Los selectores pueden ser *explícitos* o *implícitos*. Decimos que es *explícito* cuando el selector debe ser referenciado explícitamente en el código del programa al ser utilizado.

Decimos que el *selector es implícito* cuando no es necesario definir o tener un identificador para el mismo. En este caso, el elemento seleccionado de forma implícita será algún *elemento distinguido* de la estructura (es el primero, es el último, es el elemento corriente o apuntado por el cursor, etc.). Decimos que implícitamente, sin necesidad de ninguna referencia explícita, hacemos referencia a un elemento de la estructura.

## 6.6 TIPO BASE

Las estructuras de datos pueden almacenar elementos o datos y estos pertenecerán a un tipo de dato. Sin embargo una estructura puede almacenar no solo valores de un único tipo de dato sino de distintos tipos de datos. A su vez, los elementos de la estructura pueden ser valores simples o elementales (tales como el tipo carácter o entero, etc.) o también podrían ser de un tipo compuesto o estructurado, es decir que los elementos de una estructura de datos pueden a su vez ser estructuras de datos.

Entonces, el *tipo de dato base* de una estructura, que es el tipo de dato de sus elementos, puede ser:

*Simple*: los elementos no son a su vez estructuras de datos.

*Compuesto*: los elementos son a su vez estructuras de datos.

A su vez, el tipo de dato base de una estructura, que es el tipo de dato de sus elementos, también puede ser:

*Homogéneo*: los elementos son todos del mismo tipo. Es decir que los valores, que se almacenan en los elementos de la estructura, pertenecen a un único tipo de dato.

*Heterogéneo*: los elementos pueden no ser todos del mismo tipo. Es decir que los valores, que se almacenan en los elementos de la estructura, pueden pertenecer a diversos tipos de dato.

# 7 ESTRUCTURAS DE DATOS: ARREGLOS Y REGISTROS

## 7.1 ARREGLOS

Un *arreglo*, es una estructura de datos que permite almacenar elementos cuyos valores son todos del mismo tipo, llamado el *tipo base del arreglo*. Es decir que el tipo base del arreglo es homogéneo.

Que los elementos sean del mismo tipo no quiere decir que tengan el mismo valor. En efecto, aún cuando los elementos puedan guardar valores del mismo tipo no necesariamente tendrán el mismo valor: dependerá de lo que se quiera almacenar.

Una de las principales características de los arreglos es que sus elementos se encuentran *contiguos* en la memoria, es decir uno al lado del otro, y se *accede* a ellos a través de lo que se llama un *índice*.

Por estar contiguos se los puede contar a partir de un elemento que puede señalarse como el primero. El índice sirve, entonces, para *señalar*, *seleccionar*, o *indicar* o *indexar* a los distintos elementos, y es el *selector* de los elementos del arreglo. Es decir, el arreglo tiene un selector explícito: su índice.

Así como los elementos del arreglo tienen un tipo base, el índice del arreglo también tiene un tipo: el *tipo del índice*. Este debe tener la particularidad de que sus valores sean correlativos, es decir debe ser posible establecer una relación biunívoca con un subconjunto de los naturales.

En general, en los lenguajes de programación, existen tres maneras de indexar los elementos de un arreglo:

*Indexación basada en cero*: El primer elemento del arreglo es indexado por el índice `0`.

*Indexación basada en uno*: El primer elemento del arreglo es indexado por el índice `1`.

*Indexación basada en un número n*: El índice base del arreglo puede ser elegido libremente. Algunos lenguajes de programación permiten este tipo de indexación, también permiten valores de índices negativos y otros tipos de datos escalares como las enumeraciones, o los caracteres.

Es posible por medio del índice, debido a la disposición contigua de sus elementos, acceder, inspeccionar un elemento cualquiera del arreglo, sin necesidad de acceder a otros antes, es decir, puede hacerse de forma directa y aleatoria. Siempre se puede saber donde se encuentra un elemento con relación a los otros, por lo que el índice que lo identifica se puede calcular por medio de una expresión.

Los arreglos pueden tener múltiples dimensiones, por lo que no es extraño que se tenga que usar más de un índice para acceder a sus elementos; en realidad, habrán tantos índices como dimensiones tenga el arreglo. Así, el número de índices necesarios para acceder a un elemento del arreglo se llama *dimensión* o *dimensionalidad* o *rango* del arreglo. Los arreglos con dos dimensiones o bidimensionales son muchas veces referidos como *matrices*.

### ORDEN CRONOLÓGICO

Por las características de esta estructura, no puede reconocerse un orden cronológico de llegada, salida, recorrida o inspección. De hecho, los elementos de un arreglo pueden ser *accedidos* en cualquier orden.

### CAPACIDAD

Lo habitual es que la capacidad de un arreglo sea *estática*, es decir, no crezca ni disminuya con las inserciones y supresiones. En este caso, el arreglo ya tiene una capacidad establecida y no cambia.

Sin embargo, algunos lenguaje de programación permiten cambiar la capacidad de los arreglos a lo largo de la ejecución del programa. Este tipo de arreglos se los conoce como *arreglos dinámicos o extensibles*.

### OPERACIONES

Cuando estamos trabajando con arreglos estáticos, dado que su tamaño ya está establecido desde su declaración y no cambia, no será posible agregar nuevos elementos sino cambiar los valores de los elementos ya existentes. En consecuencia, hablaremos de *asignaciones* de valores a los elementos. Así, las operaciones que pueden realizarse sobre un arreglo estático son, por un lado, la de *asignar* un nuevo valor en un lugar señalado por el índice y, por otro, la de *inspeccionar* una posición del arreglo en un lugar señalado por el índice.

### REPRESENTACIÓN GRÁFICA

La Figura 7.1 muestra una posible representación gráfica de un arreglo en un lenguaje hipotético cuyo tipo base es carácter (los valores almacenados en el mismo serán todos de tipo carácter), es decir un arreglo de caracteres, y cuyo índice son enteros entre `1` y `5`.

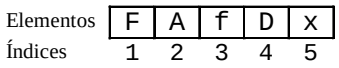


**Figura 7.1. Representación gráfica en un lenguaje hipotético de un arreglo de caracteres de 5 posiciones**

## 7.2 LOS ARREGLOS EN C

Los arreglos en C son estructuras de datos estáticas. C no soporta arreglos dinámicos, es decir, los arreglos en C mantienen su tamaño en memoria a lo largo de todo su tiempo de vida.

Las reglas de alcance de los arreglos son las mismas que las de cualquier identificador: podrán ser declarados locales a una función, con una permanencia automática como cualquier variable local, o bien globales en cuyo caso su permanencia será estática, como cualquier variable global. Asimismo, los arreglos locales también pueden ser calificados con el calificador `static`, en caso de ser necesario.

En C, las posiciones de los elementos de un arreglo siempre comienzan a contarse desde `0`, por lo que los índices del arreglo siempre serán enteros o expresiones que evalúen a valores enteros mayores o iguales a `0`.

Por ejemplo, la Figura 7.2 muestra un arreglo de nombre `a` de `10` posiciones enteras. Cada elemento del arreglo puede ser accedido con el nombre del arreglo, seguido de la posición entre corchetes (`[]`). Así, la primera posición será `a[0]`, la segunda `a[1]`, la tercera `a[2]` y así sucesivamente. En general, la posición `i`-ésima será `a[i-1]`.

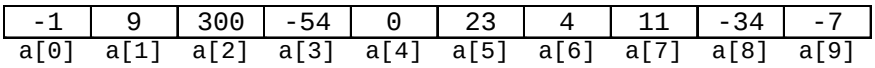


**Figura 7.2. Arreglo a de 10 elementos en C.**

Suponiendo que la variable `x` vale `2`, la sentencia de asignación

```
a[2+x] = a[2+x] + x;
```

Le sumará `2` a la quinta posición de `a` (`a[4]`).

### DECLARACIONES DE ARREGLOS

Los arreglos ocupan espacio en la memoria. La cantidad de espacio que ocupen va a depender de su tamaño y del tipo base, es decir, del tipo de sus elementos. El compilador reserva el espacio para un arreglo cuando el mismo es declarado. Así, por ejemplo, la declaración en C para el arreglo de la Figura 7.2 es:

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

También es posible declarar varios arreglos del mismo tipo base en una única línea:

```
int b[50], c[8];
```

Dado que los índices del arreglo comienzan en `0`, la última posición será siempre igual al tamaño del arreglo menos `1`.

Los arreglos pueden ser inicializados en su declaración indicando de forma explícita los valores que se quieren almacenar. Por ejemplo,

```
int a[4] = {20, 30, 40, 10};
char b[10] = {'a', 'B', '?'};
float c[] = {3.1, 4.5, 2.6};
```

`a` es un arreglo de `4` elementos enteros inicializados con los valores `20`, `30`, `40` y `10` en las posiciones `0` a `3`, respectivamente.

El arreglo `b` es un arreglo de `10` caracteres, donde sólo las posiciones `0`, `1` y `2` han sido inicializadas con los caracteres `'a'`, `'B'` y `'?'`, respectivamente. Las restantes 7 posiciones son automáticamente puestas en 0, es decir el carácter nulo (NUL), que en C se representa `'\0'`. Esto ocurre siempre cuando se inicializan sólo algunas de las posiciones de un arreglo en su declaración. Sin embargo, es importante recordar que no sucede lo mismo con aquellos arreglos que son declarados sin inicialización: las posiciones que se reservan contienen 'basura', es decir, cualquier valor.

`c` es un arreglo con `3` componentes flotantes que han sido inicializadas con los valores `3.1`, `4.5` y `2.6`. Puesto que el tamaño del arreglo no ha sido dado en la declaración, el compilador lo toma de la cantidad de datos presentes en la lista de inicializadores.

## EJEMPLO DE USO

El programa de la Figura 7.3, cuya salida es mostrada en la Figura 7.4, utiliza un arreglo `a` de `10` enteros al cual lo inicializa en `0` y luego muestra sus posiciones en un formato de tabla.

```
#include <stdio.h>
#define SIZE  10

int main()
{
  int a[SIZE], i;

  for (i = 0 ; i < SIZE; i++)
      a[i] = 0;

  printf("Indice\tElemento\n");

  for (i = 0 ; i < SIZE; i++)
      printf("%d\t%d\n", i, a[i]);

   return 0;
}
```

**Figura 7.3. Programa que inicializa un arreglo en 0 y lo muestra por pantalla.**

En la línea `#define SIZE 10` se define la constante simbólica `SIZE` cuyo valor es `10`. Una constante

simbólica es un identificador que será sustituido con el texto de sustitución (en este caso `10`) por el preprocesador de C antes de que el programa sea compilado. El preprocesador reemplazará todas las ocurrencias de la constante simbólica `SIZE` por `10` (el texto de sustitución).

```
Indice  Elemento
0       0
1       0
2       0
3       0
4       0
5       0
6       0
7       0
8       0
9       0
```

**Figura 7.4. Salida del programa de la Figura 7.3.**

## ARREGLOS MULTIDIMENSIONALES

Al igual que muchos otros lenguajes imperativos, C admite el uso de arreglos multidimensionales. En la Figura 7.5 se muestra un ejemplo de un arreglo `a` de dos dimensiones (matriz), cuya declaración es la siguiente

```
int a[2][3];
```

`a` contiene `2` filas y `3` columnas. Para hacer referencia a un elemento de un arreglo multidimensional se coloca su nombre seguido de los índices, por ejemplo el elemento `a[1][2]` hace referencia a la segunda fila tercera columna.

| | *Columna* 0 | *Columna* 1 | *Columna* 2 |
|---|---|---|---|
| *Fila* 0 | a[0][0] | a[0][1] | a[0][2] |
| *Fila* 1 | a[1][0] | a[1][1] | a[1][2] |

**Figura 7.5. Matriz a de 2 filas y 3 columnas.**

Al igual que los arreglos unidimensionales, los arreglos multidimensionales pueden ser inicializados en su declaración. Por ejemplo,

```
int a[2][3] = {{3, 8, 4}, {-3, 6, 1}};
```

declarará la matriz `a` y la inicializará agrupando los valores por fila como se muestra en la Figura 7.6. El mismo efecto produce la inicialización siguiente, ya que el compilador va inicializando los elementos por filas:

```
int a[2][3] = {3, 8, 4, -3, 6, 1};
```

| | *Columna* 0 | *Columna* 1 | *Columna* 2 |
|---|---|---|---|
| *Fila* 0 | 3 | 8 | 4 |
| *Fila* 1 | -3 | 6 | 1 |

**Figura 7.6. Matriz a inicializada.**

En caso de faltar elementos, el compilador los inicializa en `0`. Por ejemplo,

```
int a[2][3] = {{3, 8}, {-3, 6, 1}};
```

dejará a la matriz a como se ve en la Figura 7.7. Mientras que la declaración

```
int a[2][3] = {3, 8, 4, -3};
```

la dejará como se muestra en la Figura 7.8.

| | *Columna* 0 | *Columna* 1 | *Columna* 2 |
|---|---|---|---|
| *Fila* 0 | 3 | 8 | 0 |
| *Fila* 1 | -3 | 6 | 1 |

**Figura 7.7. Matriz a completada con ceros.**

| | *Columna* 0 | *Columna* 1 | *Columna* 2 |
|---|---|---|---|
| *Fila* 0 | 3 | 8 | 4 |
| *Fila* 1 | -3 | 0 | 0 |

**Figura 7.8. Matriz a completada con ceros.**

## RELACIÓN ENTRE ARREGLOS Y PUNTEROS EN C

En C, los arreglos y los punteros están muy relacionados. Todas las operaciones que se realizan con arreglos pueden también ser realizadas usando punteros.

Supongamos la siguiente declaración

```
int a[5] = {1, 4, -5, 0, 8};
int *p;
```

En C, el nombre de un arreglo es equivalente a la dirección base del mismo; por lo tanto, `a` es lo mismo que `&a[0]`. Así, por ejemplo, luego de la asignación

```
p = a;  /* es equivalente a p = &a[0]; */
```

el puntero `p` apunta al comienzo del arreglo `a`, como se ve en la Figura 7.9.

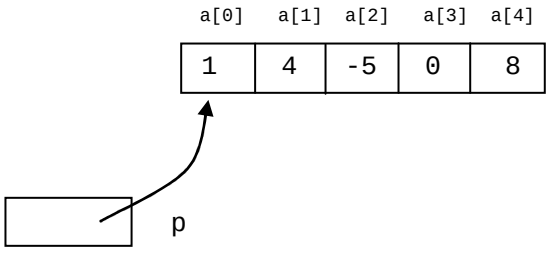


**Figura 7.9. El puntero p apunta a la dirección base del arreglo a (&a[0])**

En consecuencia, si queremos mostrar qué valor hay almacenado en la primera posición de `a` podemos hacer

```
printf("%d", a[0]);
```

o lo que es lo mismo,

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

también, el mismo resultado lo obtendremos, en este caso, haciendo

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

En C es posible realizar una gran cantidad de operaciones con los punteros, gracias a una *aritmética de punteros.*

## ARITMÉTICA DE PUNTEROS

En C, se pueden realizar algunas operaciones aritméticas con punteros. Un puntero puede ser incrementado (`++`) o decrementado (`--`), se le puede sumar o restar un valor entero, e incluso pueden sumarse y restarse punteros entre sí bajo ciertas condiciones.

Continuando con el ejemplo de la Figura 7.9, la asignación

```
p = p + 2;
```

dejará al puntero `p` apuntando al elemento `a[2]` ya que antes se encontraba apuntando a `a[0]`. En general, si `p` apunta a la componente `j` de un arreglo `y` se le suma `i` quedará apuntando `i` posiciones más adelante en el arreglo, es decir, a la posición `j+i` (Figura 7.10).

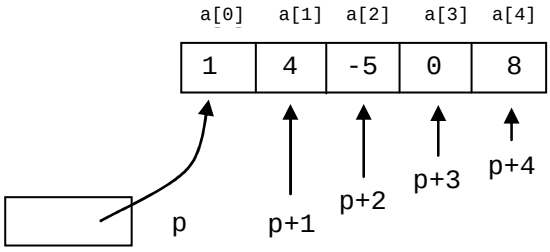


**Figura 7.10. Sumando constantes enteras a un puntero.**

Los punteros pueden ser sustraídos y sumados entre sí. Supongamos las siguientes declaraciones y asignaciones

```
int a[5] = {1, 4, -5, 0, 8};
int *p, *q, i;
p = a;
q = &a[3];
i = q – p;
```

dejará en la variable `i` el valor `3`, es decir, la cantidad de posiciones del arreglo entre `p` y `q`. Esta operación tiene sentido siempre y cuando los dos punteros se encuentren apuntando a un mismo arreglo (donde todas sus posiciones son consecutivas y del mismo tipo y en consecuencia ocupan la misma cantidad de bits).

Los punteros pueden también ser comparados entre sí, siempre que sean de un mismo tipo puntero:

```
p == q; p != q; p <= q; p < q; p >= q; p > q
```

Teniendo en cuenta lo dicho anteriormente, otras operaciones interesantes pueden ser llevadas a cabo cuando una variable puntero es igual a la posición base de un arreglo. Por ejemplo, si `p` apunta a la dirección de `a[0]`, cualquier elemento del arreglo `a` puede ser referenciado por el puntero; así, por ejemplo, `a[4]` es lo mismo que la expresión `*(p + 4)` y, en general, el elemento en la posición `i` estará dada por la expresión `*(p + i)`.

Así como un elemento de un arreglo puede ser referenciado con puntero, su dirección también puede ser obtenida con el puntero; por ejemplo, si `p == a`, es decir `p` apunta a la dirección de `a[0]`, `&a[4]` es lo mismo

que `p + 4`.

Además, los punteros, cuando apuntan a la dirección base del arreglo, pueden ser usados con suscriptos; por ejemplo, `a[1]` podría ser representado también como `p[1]`.

## PASAJE DE UN ARREGLO COMO PARÁMETRO DE UNA FUNCIÓN

En C, para pasar un arreglo como parámetro de una función basta con indicar su nombre sin colocar los corchetes. Por ejemplo, dada la declaración del arreglo `ar`

```
int ar[12];
```

la llamada a función

```
modificarArreglo(ar, 12);
```

pasa el arreglo `ar` y su dimensión a la función `modificarArreglo` (muchas veces, cuando se pasa un arreglo a una función también se pasa su tamaño).

En C, los arreglos siempre son pasados por dirección. En realidad, cuando se pasa un arreglo como parámetro lo que se está haciendo es simular un pasaje por dirección o referencia ya que, como hemos dicho, C pasa todos los parámetros por valor. Lo que realmente ocurre es que, al colocar el nombre del arreglo en la invocación y puesto que el nombre del arreglo es equivalente a la dirección base del mismo, lo que se está pasando es el valor de la dirección base del arreglo, es decir, se está pasando por valor una dirección o referencia. La invocación anterior sería equivalente a la invocación siguiente:

```
modificarArreglo(&ar[0], 12);
```

En consecuencia, cuando pasamos un arreglo como parámetro, cualquier cambio que se haga sobre el parámetro formal se está haciendo directamente sobre el parámetro real o actual.

En cuanto a la declaración del parámetro formal arreglo, se deberá colocar el tipo y nombre del arreglo; su tamaño no es necesario, de hecho el compilador lo ignora. Por ejemplo, el encabezado de la función `modificarArreglo` podría ser definido como

```
void modificarArreglo(int b[], int max) { … }
```

indicando que `modificarArreglo` acepta un arreglo de enteros en el parámetro `b` y un entero en el parámetro `max`.

Asimismo, la declaración del prototipo de la función `modificarArreglo` será:

```
void modificarArreglo(int [], int);
```

aunque también podríamos haberle dado cualquier nombre a los identificadores en el prototipo; recordemos que el compilador los ignora:

```
void modificarArreglo(int unNombreDeArreglo[], int unNombreDeVariable);
```

El programa de la Figura 7.11 muestra la diferencia entre el pasaje de todo el arreglo y el pasaje de un solo elemento del arreglo. El programa, en primer lugar, muestra los elementos del arreglo, luego se modifica el arreglo sumándole `1` a cada uno de sus elementos en la función `modificarArreglo`; a continuación se llama repetidas veces la función `modificarElemento` que recibe como parámetro un entero, donde se le pasará uno a uno los elementos del arreglo y los pondrá en `0`. Finalmente, muestra nuevamente los elementos del arreglo. Como se puede observar al ejecutar el programa (Figura 7.12), los elementos han sido modificados en la función `modificarArreglo`, sin embargo las llamadas a `modificarElemento` dejan sus elementos sin cambios ya

que fueron pasados por valor.

```
#include <stdio.h>
#define MAX     5

void muestraArreglo(int [], int);
void modificarArreglo(int [], int);
void modificarElemento(int, int);

int main()
{
    int a[MAX] = {2, 4, 6, 8, 10};
    int i;

    printf("Los valores del arreglo antes de modificarlos son:\n");
    muestraArreglo(a, MAX);
    modificarArreglo(a, MAX);
    printf("Los valores del arreglo despues de llamar"
           " a modificarArreglo son:\n");
    muestraArreglo(a, MAX);
    for (i = 0; i < MAX; i++)
        modificarElemento(a[i], i);
    printf("Los valores del arreglo despues de llamar"
           " a modificarElemento son:\n");
    muestraArreglo(a, MAX);
    return 0;
}

void muestraArreglo(int b[], int max)
{
     int i;

     for (i = 0; i < max; i++)
         printf("%d ", b[i]);
     printf("\n");
}

void modificarArreglo(int b[], int max)
{
     int i;

     for (i = 0; i < max; i++)
         b[i] = b[i] + 1;
}

void modificarElemento(int elem, int pos)
{
     elem = 0;
     printf("El valor del elemento %d dentro de"
            " modificarElemento es: %d\n", pos, elem);

}
```

**Figura 7.11. Pasaje de un arreglo y de sus elementos**

```
Los valores del arreglo antes de modificarlos son:
2 4 6 8 10
Los valores del arreglo despues de llamar a modificarArreglo son:
3 5 7 9 11
El valor del elemento 0 dentro de modificarElemento es: 0
El valor del elemento 1 dentro de modificarElemento es: 0
El valor del elemento 2 dentro de modificarElemento es: 0
El valor del elemento 3 dentro de modificarElemento es: 0
El valor del elemento 4 dentro de modificarElemento es: 0
Los valores del arreglo despues de llamar a modificarElemento son:
3 5 7 9 11
```

**Figura 7.12. Salida del programa de la Figura 7.11.**

Para lograr que la función `modificarElemento` efectivamente cambie los valores de los elementos del arreglo, el elemento a modificar debería ser pasado por referencia, por lo que en la invocación dentro del `for` debería pasarse la dirección del elemento del arreglo y no su valor, es decir,

```
modificarElemento(&a[i]);
```

Asimismo, el prototipo de la función `modificarElemento` y su definición deberían cambiarse, respectivamente, por

```
void modificarElemento(int *, int);
```

y

```
void modificarElemento(int *elem, int pos)
{
     *elem = 0;
     printf("El valor del elemento %d dentro de"
          " modificarElemento es: %d\n", pos, *elem);
}
```

Al ejecutarse el programa con los cambios introducidos, la salida producida será la mostrada en la Figura 7.13.

```
Los valores del arreglo antes de modificarlos son:
2 4 6 8 10
Valores del arreglo despues de llamar a modificarArreglo son:
3 5 7 9 11
El valor del elemento 0 dentro de modificarElemento es: 0
El valor del elemento 1 dentro de modificarElemento es: 0
El valor del elemento 2 dentro de modificarElemento es: 0
El valor del elemento 3 dentro de modificarElemento es: 0
El valor del elemento 4 dentro de modificarElemento es: 0
Valores del arreglo despues de llamar a modificarElemento son:
0 0 0 0 0
```

**Figura 7.13. Salida del programa de la Figura 7.13 modificado.**

Volvamos al ejemplo del pasaje del arreglo en la función `modificarArreglo` (Figura 7.11). Teniendo en cuenta lo dicho anteriormente sobre las equivalencias entre los punteros y los arreglos, otras maneras alternativas de definir la función `modificarArreglo`, que producen el mismo efecto, son mostradas en la Figura 7.14. Las tres

definiciones dadas de `modificarArreglo` son todas equivalentes a la dada en la Figura 7.11.

```
void modificarArreglo(int *b, int size)
{
     int i;

     for (i = 0; i < max; i++)
         b[i] = b[i] + 1;
}

void modificarArreglo(int *b, int size)
{
     int i;

     for (i = 0; i < max; i++)
         *(b + i) = *(b + i) + 1;
}

void modificarArreglo(int b[], int size)
{
     int i;

     for (i = 0; i < max; i++)
         *(b + i) = *(b + i) + 1;
}
```

**Figura 7.14. Definiciones alternativas de la función `modificarArreglo` ( Figura 7.11) que muestran las equivalencias entre punteros y arreglos.**

Los arreglos multidimensionales en C en realidad son arreglos de arreglos. Cuando un arreglo multidimensional es pasado como parámetro a una función, la declaración del parámetro de la función debe incluir el tamaño de las dimensiones, salvo la primera que no es necesaria. Así por ejemplo, dada la declaración del arreglo bidimensional `m`

```
int m[2][3] = {{3, 8, 4}, {-3, 6, 1}};
```

si el arreglo `m` se pasara a una función `f,` la declaración del parámetro formal correspondiente en `f` podría ser:

```
f(int x[][3])
```

pero también podría ser:

```
f(int x[2][3])
```

ya que el compilador ignora el valor de la primera dimensión.

En la invocación a `f`, al igual que con los arreglos unidimensionales, sólo se deberá colocar el nombre del arreglo:

```
f(a)
```

## ARREGLOS DE CARACTERES Y STRINGS

Si bien en C los arreglos pueden almacenar elementos de cualquier tipo, un tipo de dato particularmente interesante en C son los arreglos de caracteres ya que son usados como *estructura soporte* para los *strings* (cadenas de caracteres), es decir, C los usa para almacenar los strings.

Los arreglos de caracteres en C tienen muchas características particulares:

1) Pueden ser inicializados de la forma tradicional de inicializar un arreglo, es decir dando una lista de elementos

entre llaves separados por comas o en una forma mucho más sencilla, dando en la parte derecha de la inicialización directamente un string. Por ejemplo,

```
char s[] = "Programacion I";
```

es equivalente a

```
char s[] = {'P', 'r', 'o', 'g', 'r', 'a', 'm', 'a', 'c', 'i', 'o', 'n',
            ' ', 'I', '\0'};
```

Cualquiera de las dos declaraciones de `s` inicializará el arreglo `s` con los caracteres individuales del string `"Programacion I"`. El compilador calcula el tamaño del arreglo a partir del largo del string dado en la declaración y le suma uno. Esto es porque en C, para indicar el fin de un string, se utiliza un carácter especial llamado el carácter nulo, cuya representación es `'\0'` y cuyo valor numérico es `0`. En consecuencia, el tamaño de `s` no es `14` sino `15`. Por lo tanto, un arreglo de caracteres que represente un string deberá ser siempre declarado con un tamaño suficiente para contener el número de caracteres del string más el carácter nulo de terminación.

Notar además que el espacio en blanco (entre la palabra `Programacion` y la letra `I`) también es un carácter, que ocupa un lugar en el arreglo. Dado que los strings son arreglos de caracteres, podremos acceder a sus caracteres como se acceden los elementos de cualquier arreglo. Así, por ejemplo, el espacio en blanco se encuentra almacenado en `s[12]`.

2) Otra característica particular de los arreglos de caracteres es que podemos, en una sola operación de lectura, leer del teclado un string y almacenarlo directamente en un arreglo de caracteres, utilizando `scanf` junto con el string de control del formato de la entrada `"%s"`. Por ejemplo, la declaración

```
char cad[10];
```

permitirá almacenar un string de hasta 9 caracteres más el carácter nulo. La invocación

```
scanf("%s", cad);
```

leerá del teclado un string y lo almacenará en `cad`. Es responsabilidad del programador asegurarse que el arreglo tenga el tamaño suficiente para contener cualquier string que pueda ser ingresado. La función `scanf` leerá los datos hasta que encuentre el primer carácter de espacio en blanco, tabulador, salto de línea (new line) o indicador de fin de archivo, sin preocuparse de cuán grande sea el arreglo. En consecuencia, hay que ser cuidadoso ya que `scanf` puede escribir más allá de la última posición del arreglo, alterando posiciones de la memoria que no le corresponden. Para evitar esto, es conveniente usar, ya que el arreglo ha sido declarado con un tamaño de `10`, el especificador `"%9s"` en lugar de `"%s"` de manera que `scanf` lea, en este caso, sólo hasta `9` caracteres.

Notemos que en la llamada a `scanf`, a diferencia de como hemos hecho hasta ahora con otras variables con datos simples, la variable `cad` ha sido pasada sin anteponerle el operador `&`. Esto es porque, aunque si bien `scanf` espera que el parámetro sea pasado por dirección, es decir, espera la dirección en memoria de la variable, en el caso particular de los arreglos, vimos que no hacía falta anteponer el operador `&` ya que el nombre del arreglo es en realidad la dirección de su primer elemento.

3) Un vector de caracteres que represente un string (es decir, que termina con '\0') podrá ser visualizado con `printf` junto con el string de control del formato de la entrada `"%s"`. Por ejemplo, el string `cad` puede ser mostrado con la sentencia

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

Esto no significa que los elementos del arreglo no puedan ser mostrados uno por uno, como en cualquier arreglo. Por ejemplo,

```
for (int i = 0; cad[i] != 0 ; i++)
      printf("%c ", cad[i]);
```

mostrará los caracteres del string `cad` separados por espacios en blanco.

Notar que no todos los arreglos de caracteres representan strings, sólo los terminados con el carácter nulo. `printf`, al igual que `scanf`, no se preocupa por el tamaño del arreglo, muestra los caracteres hasta que encuentra un carácter nulo.

### STRINGS, ARREGLOS DE CARACTERES Y PUNTEROS A `char`

Como vimos, los arreglos de caracteres son usados en C como estructura soporte para los strings. Por otro lado, hemos visto que existe una fuerte relación entre los punteros y los arreglos.

Las siguientes declaraciones son válidas y si bien producen el mismo efecto, tienen una diferencia sutil.

```
char s1[] = "hola";
char *s2 = "hola";
```

En ambas declaraciones se va a almacenar el string `"hola"` en un arreglo de `5` posiciones; en ambos casos `s1` y `s2` apuntarán a la dirección base del correspondiente arreglo. La única diferencia está en que `s1` es un valor constante ya que representa *siempre* la dirección del elemento `0` de `s1` (recordemos que `s1` es lo mismo que `&s1[0]`) y, en consecuencia, *no puede cambiar su valor*; es decir, no podemos hacer, por ejemplo, `s1++;` mientras que sí podemos hacerlo con `s2`.

Esto hace que tampoco podamos asignar a un arreglo de caracteres una constante string, salvo en la declaración del arreglo. Por ejemplo,

```
char s1[5];
s1 = "hola";  /* Incorrecto */
```

no es válido. Esto es así porque, si fuese válido, `s1` debería apuntar a la dirección en memoria donde empieza el string `"hola"` pero ya sabemos que la dirección base de `s1` no puede cambiar. En consecuencia, cuando queremos asignar un string en un arreglo de caracteres lo que debemos hacer es copiar carácter por carácter. Notemos que existen funciones en la librería `<string.h>` del compilador C para manipular strings como, por ejemplo, las funciones `strcpy` y `strncpy` que sirven para copiar un string a otro.

Cabe notar que cuando una variable de tipo `char *` es inicializada con un literal string, como por ejemplo

```
char *s2 = "hola";
```

algunos compiladores pueden ubicar el string en lugares de la memoria donde el string no puede ser modificado. Si, por alguna razón, el literal string necesita ser luego cambiado, debería haber sido almacenado en un arreglo de caracteres para asegurar la modificabilidad en todos los sistemas.

## 7.3 REGISTROS

Los *registros* (también llamados *tuplas*, *structs*, *records*) son estructuras de datos en las que almacenamos elementos que no, necesariamente, son todos del mismo tipo, aunque podrían serlo, y obviamente, pueden no tener necesariamente el mismo valor.

Los registros nos permiten agrupar varios datos, que mantienen algún tipo de relación lógica, aunque sean de distinto tipo, permitiendo manipularlos todos juntos, usando un mismo identificador, o manipularlos cada uno por separado.

Debido a que en un registro se pueden almacenar elementos cuyos tipos pueden ser distintos uno de otros, cada

uno de los elementos de esta estructura, que se conoce con el nombre genérico de *campo*, es identificado con un *nombre de campo*, que es su *selector* y que sirve justamente para seleccionar el campo (elemento) de la estructura que se desea acceder. El registro tiene, entonces, un selector explícito.

Es posible, acceder, inspeccionar en forma directa un elemento cualquiera de un registro empleando su nombre (selector), sin necesidad de acceder a otros antes.

Así, por ejemplo, una fecha podría representarse por medio de un registro que contiene tres campos numéricos `anio`, `mes` y `dia`.

### ORDEN CRONOLÓGICO

Por las características de esta estructura, no puede reconocerse un orden cronológico de llegada, salida, recorrida o inspección.

### CAPACIDAD

La capacidad de un registro es *estática*, es decir que no crece con las inserciones ni disminuye con las supresiones. No es posible agregar elementos sino almacenar nuevos valores en los elementos ya existentes. En consecuencia, hablaremos de *asignaciones* de valores a los elementos del registro (o campos), modificándose los que ya se encontraban en él.

### OPERACIONES

Las operaciones típicas que pueden realizarse sobre los registros son las de *asignar* un valor en un lugar señalado por el nombre de campo (selector) y la de *inspeccionar* un elemento señalado por el nombre de campo (selector).

### REPRESENTACIÓN GRÁFICA

La Figura 7.15 muestra una posible representación gráfica de un registro para almacenar fechas, con sus tres campos `anio`, `mes` y `dia`.

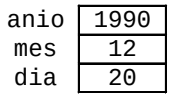


**Figura 7.15. Una representación gráfica de un registro de fechas.**

## 7.4 LOS REGISTROS EN C: STRUCTS

### DECLARACIÓN DE STRUCTS

La forma genérica para declarar un registro en C es

```
struct nombre-registro {
    tipo nombre-campo;
    tipo nombre-campo;
    ...
}
```

Por ejemplo,

```
struct fecha{
    int anio;
    int mes;
    int dia;
}
```

declara el tipo `struct fecha`.

*nombre-registro* le asigna un nombre al registro (en este ejemplo, `fecha`) el cual puede ser usado junto con la palabra clave `struct` para declarar variables de ese tipo de registro (en el ejemplo, `struct fecha`). Así, por ejemplo,

```
struct fecha fechaNacimiento, fechaCasamiento;
```

declara dos variables (`fechaNacimiento` y `fechaCasamiento`) de tipo `struct fecha`.

La declaración del tipo `struct fecha` no reserva ningún espacio en memoria, recién cuando se declaran variables es cuando se reserva el correspondiente espacio.

Los nombres de campo dentro de un registro no pueden repetirse, pero sí pueden repetirse entre registros diferentes.

En el ejemplo, en el registro `fecha` los tres campos (`anio`, `mes` y `dia`) son de tipo `int`, pero podríamos definir nuevos registros cuyos campos sean de tipos diferentes.

Las declaraciones de variables pueden venir a continuación de la declaración del tipo. Por ejemplo, la declaración

```
struct fecha{
    int anio;
    int mes;
    int dia;
} fechaNacimiento, fechaCasamiento;
```

le informa dos cosas al compilador. Por un lado, la forma y tamaño que tiene el tipo `struct` de nombre `fecha` y por otro declara las variables `fechaNacimiento` y `fechaCasamiento` de tipo `struct fecha`. Podemos, luego, definir nuevas variables usando el tipo declarado `struct fecha`, como por ejemplo

```
struct fecha fechaDefuncion, otraFecha;
```

que declara dos nuevas variables (`fechaDefuncion` y `otraFecha`) de tipo `struct fecha`.

No es obligatorio darle un nombre al registro. Directamente, se puede definir la estructura sin nombre cuando se declara una o más variables. La desventaja de esto es que no podremos definir nuevas variables del mismo tipo. Por ejemplo,

```
struct {
    int anio;
    int mes;
    int dia;
} fechaNacimiento, fechaCasamiento;
```

declara dos variables (`fechaNacimiento` y `fechaCasamiento`) de tipo `struct` con el formato especificado.

En conclusión, siempre tiene que estar presente, al menos, el nombre del `struct` o el nombre de la variable.

## OPERACIONES CON STRUCTS

Las únicas operaciones válidas que se pueden llevar a cabo con los structs son: la asignación de una variable de un tipo struct a otra otra variable del mismo tipo; obtener la dirección de un struct usando el operador de dirección (&), acceder a sus campos por medio de su selector y usar el operador `sizeof` para determinar su tamaño en bytes. Esto último será muy útil cuando trabajemos con listas encadenadas. Los structs pueden ser pasados como parámetros de una función y pueden también ser devueltos como resultado.

Los structs *no* pueden ser comparados usando los operadores `==` y `!=`.

## INICIALIZACIÓN DE UN STRUCT

Los registros en C pueden ser inicializados al igual que los arreglos en la declaración dando una lista de valores entre llaves separados por comas. Por ejemplo,

```
struct {
    int anio;
    int mes;
    int dia;
} fechaNacimiento = { 1988, 10, 5};
```

inicializará el registro `fechaNacimiento` con los valores `1988` en el campo `anio`, `10` en el campo mes y `2` en el campo `dia`. Si faltasen elementos, estos son colocados en `0`.

## ACCESO A LOS CAMPOS DE UN STRUCT

Para acceder a los campos de una variable de tipo `struct`, usaremos el nombre de la variable seguida del operador de selección de miembro de estructura (.) seguido del nombre del campo al que queremos acceder. Por ejemplo, para asignar el valor `2` al campo `dia` de la variable `fechaNacimiento` haremos

```
fechaNacimiento.dia = 2;
```

Para mostrar el contenido de la variable `fechaNacimiento` deberemos hacerlo campo a campo. Por ejemplo,

```
printf("%d/%d/%d\n", fechaNacimiento.dia, fechaNacimiento.mes,
          fechaNacimiento.anio);
```

mostrará `2/10/1988`

Los campos de un `struct` pueden ser de cualquier tipo simple o estructurado, incluso otros registros, es decir, pueden ser anidados.

## PASAJE DE UN REGISTRO COMO PARÁMETRO DE UNA FUNCIÓN

Al igual que cualquier otra variable en C, las variables de tipo `struct` son pasadas por valor. En consecuencia, para pasar un registro por dirección deberemos proceder de igual manera que antes: simular el pasaje por dirección anteponiendo el operador de dirección & al parámetro actual, declarando el parámetro formal de tipo puntero al `struct` y usando el operador `*` de desreferenciación o indirección dentro del cuerpo de la función.

El programa de la Figura 7.16 muestra el uso del tipo struct no sólo como parámetro de una función sino también como tipo del resultado de la misma. La función `modifica`, recibe como parámetro una fecha pasada por valor, la cambia dentro de la función y la devuelve como resultado, mientras que la función `muestraFecha` imprime el

valor de la fecha, que se le pase como parámetro, en el formato dd/mm/aa.

```
#include <stdio.h>

struct fecha{  /* declaracion global del tipo struct fecha */
    int anio;
    int mes;
    int dia;
};

void muestraFecha(struct fecha); /* prototipo */
struct fecha modificaFecha(struct fecha);  /* prototipo */

main(){
  struct fecha fechaNacimiento = {1988, 10, 5};

  muestraFecha(fechaNacimiento);
  fechaNacimiento = modificaFecha(fechaNacimiento);
  muestraFecha(fechaNacimiento);

  return 0;
}

void muestraFecha(struct fecha f)
{
  printf("%d/%d/%d\n", f.dia, f.mes, f.anio);
}

struct fecha modificaFecha(struct fecha f)
{
  f.dia = 29;
  f.mes = 6;
  f.anio = 2000;
  return f;
}
```

**Figura 7.16. Ejemplo de funciones que usan stucts como parámetro y como tipo de resultado.**

Al ejecutarse el programa de la Figura 7.16 , este imprimirá:

```
5/10/1988
29/6/2000
```

Queda como ejercicio, pensar en cómo se podría modificar el programa para que la función `modificaFecha`, en lugar de devolver la fecha cambiada en el valor de retorno de la función, la modifique a través del parámetro.

Un truco para pasar un arreglo por valor, es esconderlo dentro de un registro, es decir pasar un registro con un campo que contenga el arreglo. Ya que los registros son pasados por valor (a menos que simulemos el pasaje por dirección), el arreglo viene pasado dentro del registro de la misma manera. El programa de la Figura 7.17 muestra este truco. La salida del programa es mostrada en la Figura 7.18.

```
#include <stdio.h>

struct reg{
    int a[2];
};

void f(struct reg );   /* prototipo */
```

```
int main()
{
    struct reg r = {{1,2}};

    printf("Valores del arreglo a antes"
           " de llamar a f\na[0]=%d a[1]=%d\n\n", r.a[0], r.a[1]);
    f(r);
    printf("Valores del arreglo a despues"
           " de llamar a f\na[0]=%d a[1]=%d\n", r.a[0], r.a[1]);
    return 0;
}

void f(struct reg x){
     x.a[0]=0;
     x.a[1]=0;
}
```

**Figura 7.17. Arreglo pasado por valor dentro de un registro.**

```
Valores de a antes de llamar a f
a[0]=1 a[1]=2
Valores de a despues de llamar a f
a[0]=1 a[1]=2
```

**Figura 7.18. Salida del programa de la Figura 7.17.**

## USO DE `typedef`

La palabra clave `typedef` permite crear sinónimos o alias para tipos de datos ya definidos. Generalmente, se usan para abreviar los tipos structs. Por ejemplo, dada la definición del tipo

```
struct fecha{
    int anio;
    int mes;
    int dia;
};
```

la línea de programa

```
typedef struct fecha Fecha;
```

crea un sinónimo para el tipo `struct fecha` de nombre `Fecha`. A partir de este punto podemos usar en el programa el tipo `Fecha` como un sustituto más corto de `struct fecha`. Por ejemplo, podemos definir variables de tipo `Fecha:`

```
Fecha fechaNac;
```

También podemos usarlo como tipo de retorno de una función o tipo de un parámetro. Por ejemplo, el prototipo de la función `modificaFecha` podría cambiarse a:

```
Fecha modificaFecha(Fecha);  /* prototipo */
```

Podemos también usar `typedef` para definir un tipo struct en un solo paso. Por ejemplo, la definición del tipo `Fecha` que es un sinónimo de la estructura anónima siguiente

```
typedef struct {
    int anio;
    int mes;
    int dia;
} Fecha;
```

Luego podemos usar el tipo `Fecha` como lo hicimos antes.

`typedef` puede usarse con cualquier otro tipo, no solo con structs. Por ejemplo,

```
typedef int ArrInt[10];
```

define el tipo `ArrInt` como un arreglo de 10 posiciones enteras. Luego podemos usar el tipo `ArrInt`, por ejemplo para declarar la variable `ar`:

```
ArrInt ar;
```

## PUNTEROS A STRUCTS

En C, un puntero puede apuntar a cualquier tipo de dato, en particular podemos definir punteros a structs. Por ejemplo, dada la declaración del tipo `struct triangulo` y de la variable `triang`:

```
struct triangulo {
    float base;
    float altura;
} triang = {5.0, 10.0};
```

podemos declarar un puntero a `struct triangulo`

```
struct triangulo *pTriang;
```

y hacerlo apuntar a la estructura `triang`

```
pTriang = &triang
```

cuyo efecto se ve en la Figura 7.19.

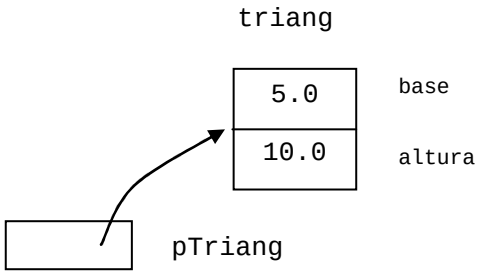


**Figura 7.19. Puntero `pTriang` apuntando al registro `triang`.**

Para acceder a los campos del registro a través del puntero, lo que hacemos es usar el operador `*` de desreferenciación que nos devolverá todo el registro y luego usar el selector de campo. Por ejemplo, para asignar `6.0` al campo `base` de `triang` a través de `pTriang` haremos

```
(*pTriang).base = 6.0;
```

El mismo efecto puede también ser logrado por medio del operador `->`, el cual lleva a cabo el mismo comportamiento que los operadores `*` y `.` combinados, pudiendo expresar lo mismo de la siguiente manera:

```
pTriang->base = 6.0;
```

# 8 ESTRUCTURAS DE DATOS: PILAS Y FILAS

## 8.1 PILAS

En una *pila* o *stack* almacenamos elementos, que son todos del mismo tipo, aun cuando, obviamente, puedan no tener necesariamente el mismo valor. Es decir, se trata de una estructura cuyo tipo base es homogéneo.

La pila es una estructura que es fácilmente identificable ya que comparte muchas de las características de las pilas que se encuentran en la vida diaria como por ejemplo una pila de libros, una pila de platos, etc.

### ORDEN CRONOLÓGICO

En una pila, el orden de llegada o *inserción* de los elementos es inverso al de salida o *supresión* de los mismos. Es decir que el primer elemento insertado será el último en ser suprimido y que el último insertado será el primero en ser suprimido.

Esto es así si se piensa en una pila de platos, el primero que se apiló es el que está abajo de todo en la pila, en la *base* de la pila y solo se lo podrá sacar después de sacar, antes, todos los que están sobre él, es decir será el que saldrá último de la pila. Por el contrario, el último que se apiló, es el que está en el *tope* de la pila y el primero que se podrá sacar de la misma.

Este orden de los elementos de una pila puede expresarse a través de las siglas en inglés de las operaciones y su orden, de la siguiente manera: **LIFO (L**ast **I**n **F**irst **O**ut).

### CAPACIDAD

La capacidad teórica de las pilas es *dinámica*, es decir que las pilas crecen con las inserciones y disminuyen con las supresiones.

### OPERACIONES

Se pueden realizar las operaciones de:

*Inserción:* solamente en el *tope* de la pila, es decir sobre el último elemento ingresado. En inglés, esta operación es conocida con el nombre de *push*.

*Supresión:* solamente del *tope* de la pila, es decir del último elemento ingresado. En inglés, esta operación es conocida con el nombre de *pop*.

*Inspección:* solamente del *tope* de la pila, es decir del último elemento ingresado.

Es importante remarcar el hecho que en la pila solo puede operarse sobre un elemento distinguido: el que se encuentra en el *tope* de la misma, es decir sobre aquél elemento que ingresó último y que es el primero que se encuentra disponible para salir. La pila tiene, entonces, un *selector implícito*, ya que es el elemento que se encuentra en el tope sobre el que se puede operar. No hay necesidad de ninguna referencia explícita al elemento, ya que las operaciones se hacen siempre sobre la misma posición de la pila: el tope.

Dado que las pilas son estructuras dinámicas, es necesario además contar dos con predicados lógicos para *controlar el desborde* y el *desfonde* de la estructura.

### REPRESENTACIÓN GRÁFICA

Por supuesto la representación gráfica puede hacerse como se desee, verticalmente en el papel, con el tope arriba o abajo; horizontalmente en el papel, con el tope a la derecha o a la izquierda; o inclusive oblicuamente. En el caso de la Figura 8.1, la pila está representada verticalmente, con el tope arriba. En la pila de la Figura 8.1(a), el orden en que llegaron los elementos (fueron insertados en la pila) fue `'f'`, `'R'`, `'+'`, `'Y'`, `'z'`, quedando `'z'` en el tope. El primer elemento a ser suprimido será, en consecuencia, `'z'`, como se ve en la Figura 8.1(b).

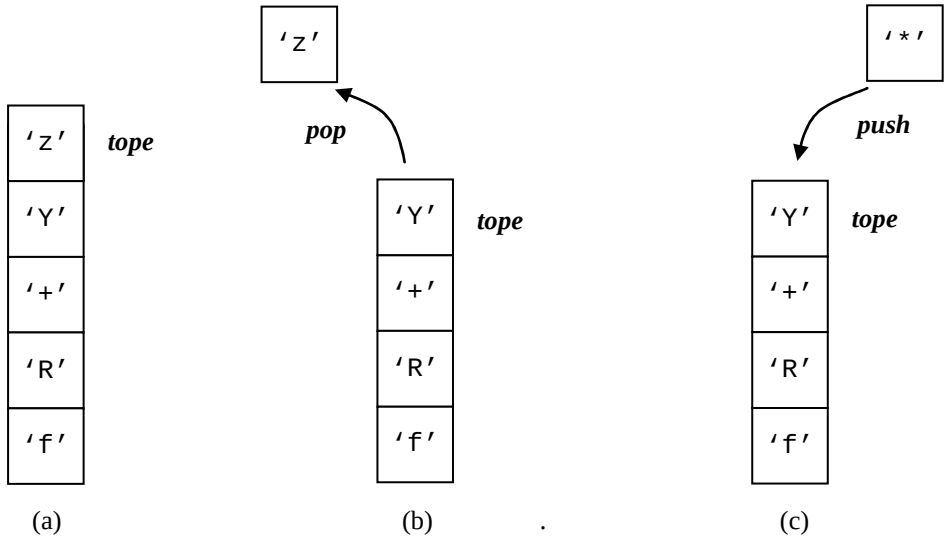


**Figura 8.1. Una pila de caracteres**

## 8.2 FILAS O COLAS

En una fila o cola almacenamos elementos que son todos del mismo tipo aun cuando, obviamente, puedan no tener necesariamente el mismo valor. Se trata, en consecuencia, de una estructura cuyo tipo base es homogéneo.

La fila es una estructura que es fácilmente identificable ya que comparte muchas de las características de las filas o colas que se encuentran en la vida diaria, como por ejemplo una cola de personas esperando el colectivo, una cola en el cajero del supermercado, etc.

### ORDEN CRONOLÓGICO

En una fila, el orden de llegada o *inserción* de los elementos es el mismo que el de salida o *supresión* de los mismos. Es decir que el primer elemento insertado será, también, el primero en ser suprimido.

Esto es así, si pensamos en una cola de personas esperando, para ser atendidas, frente a una ventanilla de un cajero en un banco (¡siempre y cuando las personas sean respetuosas del orden y no 'salten' lugares en la cola!). La primera persona en llegar (inserción) será la que está primera en la cola y será la primera en ser atendida (supresión).

De ahí que en las filas o colas sea posible hablar de dos elementos distinguidos: el *primero* y el *último* elemento. Es precisamente sobre estos elementos distinguidos donde podemos operar con una fila.

Este orden de los elementos de una fila puede expresarse a través de las siglas en inglés de las operaciones y su orden, de la siguiente manera: **F**irst **I**n **F**irst **O**ut (**FIFO**).

## CAPACIDAD

La capacidad teórica de las colas es *dinámica*, es decir que las filas crecen, con las inserciones y disminuyen con las supresiones.

## OPERACIONES

Se pueden realizar las operaciones de:

*Inserción*, después del *último* elemento ingresado.

*Supresión*, del *primer* elemento ingresado.

*Inspección*, del *primer* elemento ingresado.

Es importante remarcar el hecho que en las filas sólo puede suprimirse e inspeccionarse el primer elemento ingresado y solo se inserta un nuevo elemento después del último elemento ingresado.

La fila tiene dos selectores implícitos, ya que es el elemento que se encuentra en el primer lugar de la misma sobre el que se puede operar para eliminarlo o inspeccionarlo y se puede insertar sólo después del elemento que se encuentra en el último lugar. No hay necesidad de ninguna referencia explícita a un elemento, ya que las operaciones se hacen siempre sobre alguna de dos elementos distinguidos de la fila: o el *primero* o después del *último*.

Como las filas son estructuras dinámicas, es necesario además contar dos con predicados lógicos para *controlar el desborde* y el *desfonde* de la estructura.

## REPRESENTACIÓN GRÁFICA

La fila de la Figura 8.2 está representada gráficamente de derecha a izquierda, en este caso el orden de llegada (inserción) de los elementos fue: `'Z'`, `'x'`, `'T'`, `'*'`, `'F'`; es decir `'Z'` fue el primero en llegar (insertado) y`'F'`el último en llegar (insertado). Consecuentemente `'Z'` será el primero en salir (suprimido) y `'F'` será el último en salir (suprimido).

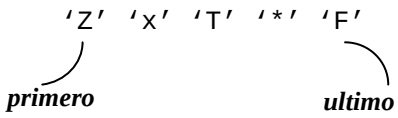


**Figura 8.2. Una fila de caracteres.**

Por supuesto, la representación gráfica puede hacerse como se desee, verticalmente en el papel, con el primero arriba o abajo; horizontalmente en el papel, con el primero a la derecha o a la izquierda; o inclusive oblicuamente.

# 9 ESTRUCTURAS DE DATOS: LISTAS UNI Y BIDIRECCIONALES

## 9.1 LISTAS UNIDIRECCIONALES

### COMPOSICIÓN DE LOS ELEMENTOS

Los elementos de una lista unidireccional o secuencia, llamados *nodos*, constan de *dos partes*:

La ***V**ariable de **I**nformación **P**ropiamente **D**icha* (*VIPD*), que es donde se almacena la información que corresponde a cada elemento de la lista. Las VIPDs de una lista son todas del mismo tipo, aun cuando, obviamente, puedan no tener necesariamente el mismo valor.

El *puntero* al elemento (nodo) siguiente en la lista, el cual puede encontrase no necesariamente contiguo. En el caso del último elemento, el puntero no apunta a un elemento y se dice que su valor es *nil*.

### ORDEN CRONOLÓGICO

Las listas tienen un orden para la inspección de sus elementos, que va del primer elemento al último. Es decir, para acceder al elemento `i`-ésimo de la lista es necesario haber pasado por los `i-1` elementos anteriores. Este orden está dado por los punteros de cada elemento, donde cada uno apunta al elemento que le sigue en la secuencia. Para ello se cuenta con un *acceso inicial* que apunta al *primer* elemento de la lista. En caso de que la lista se encuentre vacía, no habrá primer elemento y el acceso apuntará a *nil*.

No hay un orden cronológico ni para la entrada (inserción) de elementos ni para la salida (supresión) de los mismos. Es decir, los elementos pueden insertarse y suprimirse en cualquier orden.

### CAPACIDAD

La capacidad teórica de las listas unidireccionales es *dinámica*, es decir que crece con las inserciones y disminuye con las supresiones.

### OPERACIONES

Se pueden realizar las operaciones de:

*Inserción* de un elemento, en cualquier lugar de la estructura.

*Supresión* de cualquier elemento que se encuentre en la estructura.

*Inspección* de cualquier elemento que se encuentre en la estructura.

Dado que las listas son estructuras dinámicas, es necesario además contar con dos predicados lógicos para *controlar el desborde* y el *desfonde* de la estructura.

En todos los casos, el acceso es secuencial, es decir del primero al último elemento. Quiere decir que, por ejemplo, para acceder (inspeccionar) el quinto elemento, a partir del primero, debe pasarse por los cuatro elementos anteriores. Esto suponiendo que la lista tenga al menos cinco elementos.

Las listas unidireccionales tienen asociado un *cursor* que permite marcar, señalar, un elemento. Este elemento, señalado o apuntado por el cursor, será llamado el *elemento corriente*; es el elemento sobre el cual se puede operar, es decir que puede inspeccionarse, suprimirse, o donde, entre él y el anterior, podrá insertarse un nuevo elemento.

Para mover el cursor dentro de la lista, será necesaria, además, una operación para *avanzar* el cursor de a una

posición, y otra para *colocarlo en la primera posición* de la lista.

Un cursor es básicamente un puntero externo a la lista que toma un valor de puntero, es decir, apunta a un elemento de la lista o inclusive a *nil*. Podemos decir que el cursor de una lista es el *selector implícito* de la misma.

### REPRESENTACIÓN GRÁFICA

La lista de la Figura 9.1 está representada gráficamente de izquierda a derecha. Los nodos marcados como *primer elemento* y *último elemento* se corresponden con el orden de acceso o inspección de los elementos de la lista, considerado desde el acceso a la misma.

Por supuesto, la representación gráfica puede hacerse como se desee, verticalmente en el papel, con el *primero* arriba o abajo; horizontalmente en el papel, con el *primero* a la derecha o a la izquierda; o inclusive oblicuamente.

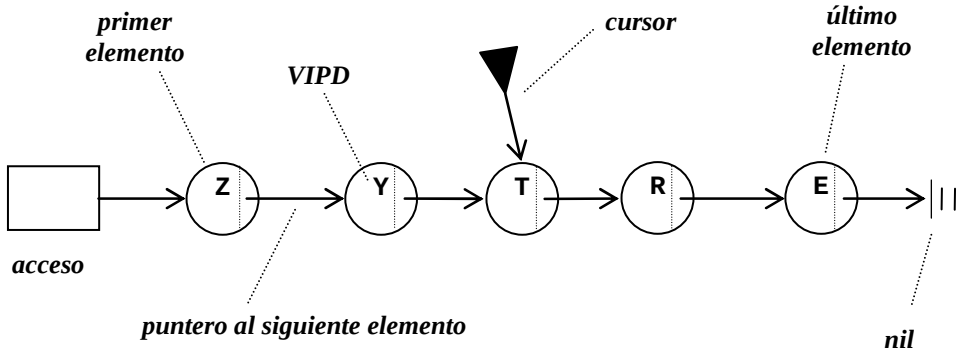


**Figura 9.1 Una lista de caracteres.**

## 9.2 LISTAS BIDIRECCIONALES

### COMPOSICIÓN DE LOS ELEMENTOS

Los elementos o nodos de una lista bidireccional constan de *tres partes*:

La ***V**ariable de **I**nformación **P**ropiamente **D**icha* (*VIPD*), que es donde se almacena la información que corresponde a cada elemento de la lista. Las VIPDs de una lista son todas del mismo tipo, aun cuando, obviamente, puedan no tener necesariamente el mismo valor.

El *Puntero al elemento siguiente*; este puede apuntar a *nil* cuando no hay un elemento al que apuntar (caso del último elemento). Este puntero se conoce también como puntero adelante o en inglés *forward pointer*.

El *Puntero al elemento anterior*, este puede apuntar a *nil* cuando no hay un elemento al que apuntar (caso del primer elemento). Este puntero se conoce también como puntero atrás o en Inglés *backward pointer*.

Recordemos que, como ya dijimos para las listas unidireccionales, cuando un puntero no apunta a un elemento se dice que su valor es *nil*.

### ORDEN CRONOLÓGICO

En las listas bidireccionales, al igual que en las unidireccionales, existe un orden de inspección secuencial de los elementos; sin embargo, en las bidireccionales, como su nombre lo indica, no solo va del primer elemento al último sino también del último al primero. Este orden está dado por los punteros de cada nodo de la lista, los que apuntan uno al elemento anterior y el otro al siguiente.

Igual que en el caso de las listas unidireccionales, se cuenta con un *acceso* a la lista que apunta a un elemento de la lista. Es por ello que se puede reconocer un elemento como el *primero*: es aquel que es apuntado por el acceso a la lista. Cuando la lista se encuentra vacía el acceso apunta a *nil*.

No hay un orden cronológico ni para la entrada (inserción) de elementos ni para la salida (supresión) de los mismos.

## CAPACIDAD

La capacidad teórica de las listas bidireccionales es *dinámica*, crece y disminuye con las inserciones y supresiones, respectivamente.

## OPERACIONES

Se pueden realizar las operaciones de:

*Inserción* de un elemento, en cualquier lugar de la estructura.

*Supresión* de cualquier elemento que se encuentre en la estructura.

*Inspección* de cualquier elemento que se encuentre en la estructura.

Dado que las listas son estructuras dinámicas, es necesario además contar dos con predicados lógicos para *controlar el desborde* y el *desfonde* de la estructura.

En todos los casos, el acceso es secuencial, es decir del primero al último y/o del último al primero. Quiere decir que, por ejemplo, para acceder (inspeccionar) al octavo elemento, a partir del primero, deberá pasarse por los siete elementos anteriores. Para ir al cuarto elemento, a partir del último, deberá pasarse por todos los elementos de la lista que estén entre el último y el cuarto elemento; por ejemplo, suponiendo que la lista tiene ocho elementos, significa que habrá que pasar por los elementos octavo, séptimo, sexto y quinto.

Las listas bidireccionales tienen asociado un *cursor* que permite marcar, señalar, apuntar un elemento. Este elemento, así apuntado o señalado, será el *elemento corriente*, aquel que puede inspeccionarse, suprimirse, o donde, entre el anterior y el corriente, podrá insertarse uno nuevo. En consecuencia, para poder operar sobre un elemento es necesario que el cursor se encuentre apuntándolo, es decir, que sea el elemento corriente. Si el cursor se encuentra apuntando a otro elemento, habrá que moverlo hacia delante o hacia atrás de acuerdo a dónde se lo quiera llevar. Consecuentemente, serán necesarias dos operaciones: una para *avanzar el cursor hacia adelante* de a una posición y otra para *volverlo una posición hacia atrás*. Asimismo, será necesaria otra operación para *colocarlo en la primera posición* de la lista, cuando sea necesario.

Un cursor es básicamente un puntero a la lista, que toma un valor puntero lo que le permite apuntar a un elemento de la lista, o inclusive a *nil* en situaciones tales como: cuando la lista se encuentra vacía, cuando el cursor se encuentra apuntando fuera de la lista ya sea después del último en las listas uni y bidireccionales o cuando se lo ha hecho retroceder 'después' del primero en las listas bidireccionales. Para estas situaciones existe un predicado `isOOS` (**is O**ut **o**f **S**tructure) que devolverá *verdadero* de encontrarse el cursor con el valor *nil* y *falso* en todos los otros casos.

Podemos decir que el cursor de una lista es el *selector implícito* de la misma.

## REPRESENTACIÓN GRÁFICA

La lista bidireccional de la Figura 9.2 está representada gráficamente de izquierda a derecha. Los elementos marcados como *primer elemento* y *último elemento* corresponden al orden de los elementos en la lista considerando

el orden de inspección de los mismos desde el acceso a la misma.

Por supuesto, la representación gráfica puede hacerse como se desee, verticalmente en el papel, con el primero arriba o abajo; horizontalmente en el papel, con el primero a la derecha o a la izquierda; o inclusive oblicuamente.

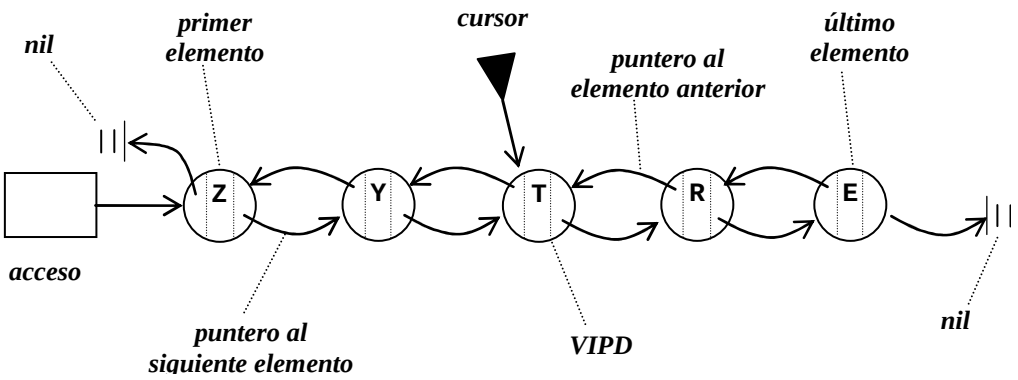


**Figura 9.2. Una lista bidireccional de caracteres**

## 9.3 GENERALIDAD DE LAS LISTAS

Las listas, uni y bidireccionales, son estructuras de datos, que por las escasas restricciones impuestas a las mismas, frente a las otras estructuras, permiten emplearlas fácilmente para representar con ellas cualquier otra estructura.

Por ejemplo, diremos que una pila no es más que una lista con restricciones en las operaciones de inserción, supresión e inspección, las que en lugar de hacerse con cualquier elemento solo pueden hacerse con el primero de la lista (que sería el tope de la pila).

Esto no quiere decir que no puedan representarse unas estructuras con otras aparte de las listas, por ejemplo filas empleando pilas, sino que las listas permiten realizar esto más fácilmente.

La Tabla 9.1 muestra un cuadro sinóptico con las características de las estructuras de datos presentadas.

**Tabla 9.1 Características de las estructuras de datos**

| **Características** | | **Estructura** | | | | |
|---|---|---|---|---|---|---|
| | | **Arreglo** | **Registro** | **Pila** | **Fila** | **Lista** |
| **Capacidad** | | Estática | Estática | Dinámica | Dinámica | Dinámica |
| **Tipo Base** | | Homogéneo | Heterogéneo | Homogéneo | Homogéneo | Homogéneo |
| **Selector** | | Explícito (Índice) | Explícito (Nombre de campo) | Implícito (Tope) | Implícito (Primero y último) | Implícito (Cursor) |
| **Orden Cronológico** | **Inspección** | No | No | Si (LIFO) | Si (FIFO) | Si (Del primero al último y del último al primero, en el caso de las bidireccionales) |
| | **Entrada** | | No | Si (LIFO) | Si (FIFO) | No |
| | **Salida** | | No | Si (LIFO) | Si (FIFO) | No |
| **Operaciones** | **Inspección** | Usando índice(s) | Usando nombre de campo | Copia | Copia | Copia |
| | **Entrada** | Asignación | Asignación | Inserción | Inserción | Inserción |
| | **Salida** | Asignación | Asignación | Supresión | Supresión | Supresión |

# 10 ESTRUCTURAS DE DATOS: ESTRUCTURAS MULTINIVEL

Las estructuras multinivel son estructuras de datos que tienen más de un nivel, es decir que cada elemento de la misma tendrá, a su vez, una estructura de datos asociada al elemento.

En este caso hablaremos de *niveles* o de *dimensiones*. Aceptaremos que para cada elemento de la estructura puede haber, a su vez, otra estructura, que sería en ese caso el segundo nivel. A su vez, este segundo nivel puede, también, tener asociado a él, en cada uno de sus elementos, una estructura de datos y así sucesivamente, potencialmente de una manera infinita.

Este *anidamiento* de estructuras puede ser tan grande como lo permita el compilador del lenguaje. Potencialmente o idealmente podría ser infinito, pero ya sabemos que en la realidad esto no es así. Diremos que el primer nivel (o dimensión) es aquel más externo en el anidamiento, es decir aquel que, partiendo de la definición, tiene en cada uno de sus elementos a su vez una estructura. Este sería el segundo. El tercero (si existe) sería aquel que constituye cada uno de los elementos del segundo y así sucesivamente.

El último nivel (o dimensión) será siempre un tipo de dato simple y no una estructura. Sin embargo éste último nivel no lo contamos como tal. Sólo contamos los niveles que son estructuras y no los que contienen datos simples. Así, por ejemplo, un arreglo de caracteres tiene un nivel, los elementos de tipo simple no se consideran como un nivel.

Sobre esto último, pueden haber otros criterios tal como considerar que una variable de tipo simple tiene un sólo nivel mientras que una estructura de datos cuyos elementos son de tipo simple agrega un nivel o dimensión. En nuestro caso, consideramos que el tipo simple no cuenta como nivel, sólo cuentan como nivel los tipos estructurados, así por ejemplo una pila cuyo tipo base es caracter tiene un nivel o dimensión.

Cada una de las estructuras de cada nivel podrá ser cualquiera de las estructuras dinámicas o estáticas ya vistas: pilas, filas, listas uni y bidireccionales, arreglos y registros, o incluso otras estructuras como las dicolas, los árboles, los anillos, etc. que no estudiamos en este curso.

Las estructuras de datos de multinivel pueden ser *homogéneas* o *heterogéneas*. Diremos que una estructura de datos multinivel es *homogénea* cuando todos los niveles (o dimensiones) de la misma, que contienen a su vez estructuras de datos, están constituidos por el mismo tipo de estructura de datos. Por supuesto, esta definición no incluye el último nivel, el cual ya dijimos que no es una estructura.

Por otro lado una estructura de datos multinivel es *heterogénea* cuando existe al menos un nivel (o dimensión) de la misma, cuyos elementos son estructuras de datos diferentes a la del primer nivel. Por supuesto, esta definición no incluye el último nivel, el cual ya dijimos que no es una estructura.

Así, por ejemplo, un arreglo que tiene como tipo base un arreglo de enteros (dos niveles ya que no contamos el tipo base del último arreglo que es un tipo simple: entero) es una estructura de dos niveles homogénea. La Figura 9.3 muestra esta situación. En la primera dimensión o nivel hay siete elementos, cada uno de los cuales es, a su vez, un arreglo de cuatro elementos. Es decir que es un arreglo con siete arreglos de cuatro elementos enteros cada uno.

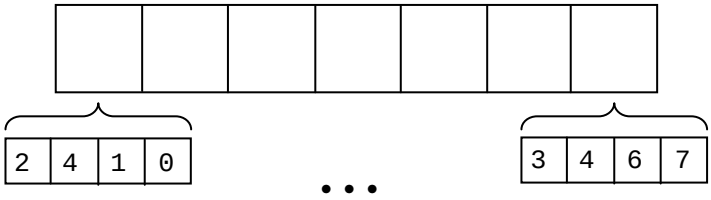


**Figura 9.3. Un arreglo de arreglos de enteros.**

Mientras que un arreglo que tiene como tipo base un registro (también dos niveles ya que no contamos el tipo

base de la última estructura –cada uno de los campos del registro– que suponemos en este ejemplo son todos tipos simples) es una estructura multinivel heterogénea. Por ejemplo, en C, esta situación podría ejemplificarse con el arreglo `personas` cuyo tipo base es `struct persona`, como sigue

```
struct persona {
     long int dni;
     char sexo;
};

struct persona personas[10];
```

## 10.1 PARTICULARIDADES DE LAS ESTRUCTURAS DE MULTINIVEL

En algunos casos, ciertas estructuras de multinivel tienen nombres especiales debido a su mayor uso. Así, cuando se tienen arreglos de multinivel homogéneos estos se conocen como arreglos multidimensionales.

Por otro lado cuando la estructura de datos no es homogénea en su tipo base, como es el caso del registro, puede darse que la misma tenga algunos de sus elementos que sean a su vez estructuras de datos mientras que otros sean tipos simples. En este caso cuando hablamos de nivel nos referiremos al mayor nivel que pueda encontrarse en la estructura.

## 10.2 ESTRUCTURAS ESTÁTICAS Y DINÁMICAS

Por sus características particulares existen diferencias en la manipulación de las estructuras multinivel entre las estáticas y las dinámicas.

Dado que en las estructuras estáticas no crece ni disminuye su tamaño cuando se ‘pone’ o ‘saca’ un elemento en las mismas, pero lo opuesto sí ocurre con las estructuras dinámicas, las operaciones tienen, como ya se ha visto, características distintas que se reflejan en la manipulación de las estructuras de multinivel.

## 10.3 OPERACIONES

Las operaciones sobre estructuras multinivel conservan las *restricciones* que corresponden a cada una de las estructuras. Esto da como consecuencia que para cada nivel, el acceso al mismo, para efectuar operaciones en ese nivel, tiene las restricciones que corresponden al nivel en cuestión.

Por ejemplo, en una pila de listas unidireccionales, la pila está en el primer nivel y cada uno de sus elementos contiene una lista unidireccional, que, en consecuencia, diremos que están en el segundo nivel. La pila permitirá acceder solamente a la lista del segundo nivel que corresponda al elemento que se encuentra en el tope de la pila. Para efectuar operaciones en las listas del segundo nivel, de los otros elementos de la pila, será necesario suprimir el tope de la pila cuantas veces sea necesario para llegar al elemento de la pila, que contiene la lista de segundo nivel a la que se desea acceder, para así poder operar sobre la lista de segundo nivel.

Así, por ejemplo, en un arreglo de arreglos (un arreglo bidimensional) para acceder a un elemento del segundo nivel será necesario emplear un índice para el primer nivel y otro índice para acceder, por fin, al elemento en el segundo nivel.

# 11 IMPLEMENTACIÓN DE ESTRUCTURAS

## 11.1 INTRODUCCIÓN

Las estructuras de datos dinámicas que hemos presentado han sido vistas hasta ahora como una "caja negra"; es decir, sabíamos que podían hacerse operaciones sobre ellas pero ignorábamos cuál era el mecanismo interno que hacía posibles dichas manipulaciones.

Dado que no todos los lenguajes de programación proveen estos tipos de datos estructurados, muchas veces, es necesario darle soporte usando los tipos de datos que sí nos brinda el lenguaje. Hablaremos entonces, indistintamente, de *simular* o *implementar* estructuras de datos.

La simulación o implementación de una estructura de datos se supone que se hará sobre una máquina, computadora, determinada, y lógicamente empleando un programa. Para hacer este programa será necesario un lenguaje y éste lenguaje deberá contar con *primitivos*, con estructuras de datos estándar, que son las que se deberán emplear para realizar la tarea de simulación o implementación.

Obviamente, si el lenguaje nos provee como primitivo la estructura que se quiere simular no hay nada que simular. En efecto, por ejemplo, si lo que se desea es usar un arreglo y el lenguaje elegido es el C, la cosa será fácil: sólo lo usaremos. De no ser así, entonces habrá que elegir la estructura que provea el lenguaje y que mejor se adapte a la implementación de la estructura que se desee simular.

La mayor parte de los lenguajes de programación tienen como estructura primitiva el arreglo, lo que permitirá que el mismo sea elegido como *estructura soporte*, es decir como estructura que permita "soportar" la estructura que se intenta simular. Esta estructura es la que nos permitirá almacenar los valores correspondientes a los elementos de la estructura simulada. Hablaremos, entonces, de dos estructuras: por un lado la estructura que queremos implementar, que es la *estructura simulada o soportada* y por otro lado la estructura que nos sirve para soportar la implementación, que es la *estructura soporte*.

Como en la mayor parte de los casos, al hacer un programa que simule una estructura no sólo se deberá elegir la estructura de datos más conveniente sino que se tendrá que considerar:

- El *tiempo* de realización del programa y el tiempo que toma el programa para ejecutarse.
- El *espacio* de almacenamiento de los datos y de almacenamiento del programa.

La eficiencia total del programa que simule una estructura será un compromiso entre el tiempo y el espacio, y por supuesto de los primitivos disponibles en el lenguaje. En este sentido, hay estructuras de datos que se prestan más fácilmente para simular otras, por ejemplo la lista o el arreglo.

Cuando se emplea un arreglo como estructura soporte, una acción que deberá realizarse, algunas veces, sobre éste es el *desplazamiento* de sus elementos. Esta es la acción de mover todos o algunos de los elementos de la estructura simulada en el arreglo, un lugar, hacia un lado u otro del arreglo.

Para ilustrar los ejemplos emplearemos el lenguaje C. Los nombres de variables y tipos empleados, así como la forma de declarar los mismos son UNA de las MUCHAS formas de encarar la solución de los problemas planteados.

## 11.2 ADMINISTRACIÓN DE LOS ESPACIOS LIBRES

Desde el momento que se implementa un mecanismo para que una estructura funcione (la estructura simulada) empleando otra (la estructura soporte), se presenta el problema de cómo emplear los espacios que tenemos disponibles en la estructura soporte para almacenar los datos que ingresarán en la estructura simulada. Sobre la

estructura soporte habrá espacios ocupados por los datos y habrá otros espacios que estarán libres, listos para ser usados para almacenar nuevos datos que ingresen en la estructura simulada.

El hecho de tener espacios libres listos para ser usados nos obliga a tener estos (que podrán ser más de uno) organizados de alguna manera, es decir mantener para los lugares libres, a su vez, una estructura de datos.

## DESBORDE Y DESFONDE

El *desborde* u *overflow* en inglés se presenta cuando, queriendo insertar un nuevo elemento en la *estructura simulada*, la *estructura soporte* se encuentra llena (sin lugares libres) y en consecuencia no puede insertarse ningún otro elemento en la *estructura simulada*.

El *desfonde* o *underflow* en inglés se presenta cuando, queriendo suprimir un elemento de la *estructura simulada*, la *estructura simulada* está vacía y en consecuencia no se puede suprimir en ella. Esta condición es propia de las *estructuras dinámicas* y corresponde al concepto de *estructura vacía*. Es decir cuando no se puede suprimir en la estructura porque ésta no tiene ningún elemento.

Por el contrario, el concepto de *desborde* es propio de la *implementación de estructuras*. La estructura que sirve de *soporte* para la estructura que se *simula o soporta* tiene una *capacidad finita* de almacenamiento. Aún cuando teóricamente la capacidad de almacenamiento de las estructuras sea infinita esto no sucede así en la práctica.

Estas condiciones de error (el desborde y el desfonde) nos obligan a su control en la implementación de estructuras. Este control puede hacerse de varias maneras:

- Empleando una variable que mantenga la cantidad de elementos que hay presentes en la estructura en todo momento. Esta variable se incrementa cada vez que se inserta y se decrementa cada vez que se suprime. La variable será *igual a la capacidad máxima* de la estructura soporte cuando haya *desborde* (no puede insertarse); y será *igual a cero* si hay *desfonde* (no puede suprimirse).
- Utilizando algunas de las características propias de la implementación (punteros, marcas, etc.). Esto se indicará más adelante en cada caso particular.

## MÉTODOS DE ADMINISTRACIÓN DE LOS ESPACIOS LIBRES

Reconocemos dos grandes métodos de administración de los espacios libres en una estructura soporte: *estático* y *dinámico*.

### ADMINISTRACIÓN ESTÁTICA

Cuando los espacios libres en la estructura soporte se administran estáticamente, cada vez que se necesita un lugar para almacenar un nuevo elemento (inserción en la estructura simulada), se usa un lugar libre de la estructura soporte. Cada vez que se libera un lugar ocupado en la estructura simulada (supresión en la estructura simulada) este NO es incorporado a la estructura de los espacios libres y, en consecuencia, NO podrá ser reusado cuando se necesite un lugar para almacenar un nuevo elemento en la estructura simulada.

Podemos comparar esto con un galpón donde se almacenan mercaderías y donde entran y salen bultos. Si inicialmente el galpón está vacío, cada vez que se desea guardar algo se ocupa un lugar. Si esto se hace con algún criterio, por ejemplo se empieza desde atrás hacia adelante y desde la derecha hacia la izquierda, el espacio se irá ocupando ordenadamente. Sin embargo cuando sale un bulto el lugar ocupado por este no es reutilizado y se siguen usando los lugares que no fueron nunca ocupados.

En este método la estructura que tenga los espacios libres deberá permitir supresiones solamente: se le saca un espacio libre para que sea empleado para una inserción en la estructura simulada. Por el contrario, cuando ocurre

una supresión en la estructura simulada, ese espacio que se libera no vuelve a los espacios libres de la estructura soporte.

Este método puede tener dos variantes:

(a) Los espacios liberados nunca más son reutilizados y, en consecuencia, cuando se terminaron los espacios libres iniciales la estructura simulada no aceptará más inserciones aún cuando haya lugares que se liberaron luego de haber estado ocupados. A esto le llamamos *administración estática permanente*.

(b) Cuando no quedan más espacios libres, puede realizarse un proceso de reordenar toda la estructura soporte recolectando los espacios que pudieron haber quedado libres para que vuelvan a ser utilizados. Este proceso se llama en inglés *garbage collection* o *recolección de basura* en español. A esta variante de administración de los espacios libres le llamamos *administración estática temporaria o períodica.*

#### ADMINISTRACIÓN DINÁMICA

Cuando los espacios libres en la estructura soporte se administran dinámicamente, cada vez que se necesita un lugar para almacenar algo (inserción en la estructura simulada) se usa un lugar libre. Cada vez que se libera un lugar ocupado en la estructura simulada (supresión en la estructura simulada) este es incorporado a la estructura de los espacios libres y en consecuencia podrá ser reusado cuando se necesite un lugar para almacenar algo en la estructura simulada.

Volviendo al ejemplo del galpón donde se almacenan mercaderías y donde entran y salen bultos. Si inicialmente el galpón está vacío, cada vez que se desea guardar algo se ocupa un lugar. Si esto se hace con algún criterio, por ejemplo se empieza desde atrás hacia adelante y desde la derecha hacia la izquierda, el espacio se irá ocupando ordenadamente. Ahora cuando sale un bulto el lugar ocupado por éste es reutilizado, es decir que se lo puede volver a ocupar si es necesario.

En este método la estructura que tenga los espacios libres debe permitir inserciones y supresiones. Supresiones cuando se le saca un espacio libre para que sea empleado para una inserción en la estructura simulada. Inserciones cuando se incorpora un espacio libre que ha quedado libre por una supresión en la estructura simulada.

Hay que señalar que, en algunos casos, la administración dinámica puede resultar automática ya que, sin realizar ninguna acción especial, una supresión en la estructura simulada implicará, de forma automática, una inserción en la estructura de los espacios libres.

### 11.3 IMPLEMENTACIÓN DE PILAS

Una forma posible de simular pilas es empleando un arreglo cuyo tipo base sea igual al de los elementos de la pila que va a simularse.

En esta sección presentamos, a modo de ejemplo, tres métodos de implementación que hacen uso de un arreglo como estructura soporte para la pila: dos con desplazamiento y uno sin desplazamiento. Los tres métodos administran los espacios libres de forma dinámica. Cabe aclarar que los métodos presentados no son los únicos. Queda al alumno pensar y analizar otras variantes.

#### CON DESPLAZAMIENTO

En este caso se fija el tope de la pila en la primera posición del arreglo y se hace un desplazamiento en el arreglo de los elementos de la pila cada vez que se hace una inserción (push) o una supresión (pop).

Será necesario tener una forma de indicar dónde está, en el arreglo, la base, es decir el final de la pila en el arreglo. Esto, a su vez, puede hacerse de dos maneras: usando una variable para la base o con una marca.

## CON VARIABLE PARA LA BASE

Con una variable que sirva de puntero para saber dónde está, en el arreglo, la base, es decir el fin, de la pila. En la Figura 11.1 se muestra una definición posible para esta estructura soporte. Se ha usado un registro para encapsular la estructura que consiste de un arreglo para almacenar los elementos de la pila (`elementos`) y un campo (`base`) que mantiene el índice del arreglo en donde se encuentra el elemento de la base de la pila.

```
#define N   100

struct pila {
   char elementos[N];
   int base;
} miPila;
```

**Figura 11.1. Definición del tipo `struct pila` usando como estructura soporte un arreglo para los elementos de la pila `miPila` y una variable `base` para indicar la base de `miPila`.**

`base` se debe incrementar o decrementar cada vez que se hace una inserción o una supresión. Esta variable puede servir para controlar el desfonde cuando es menor que `0` dado que, en C, `0` es el valor del primer índice de los arreglos y para el desborde cuando es mayor que `N-1`, ya que ese es el máximo valor del índice del arreglo `elementos`.

Los espacios libres se administran automáticamente en forma dinámica. En efecto, puede decirse que están organizados como una pila cuyo tope "toca" la base de la pila simulada, de manera tal que cada vez que inserta un elemento en la pila simulada se produce automáticamente una supresión en la pila de los espacios libres y viceversa. La variable que indica dónde está la base de la pila simulada, permite calcular al mismo tiempo dónde se encuentra el tope de la pila de los espacios libres.

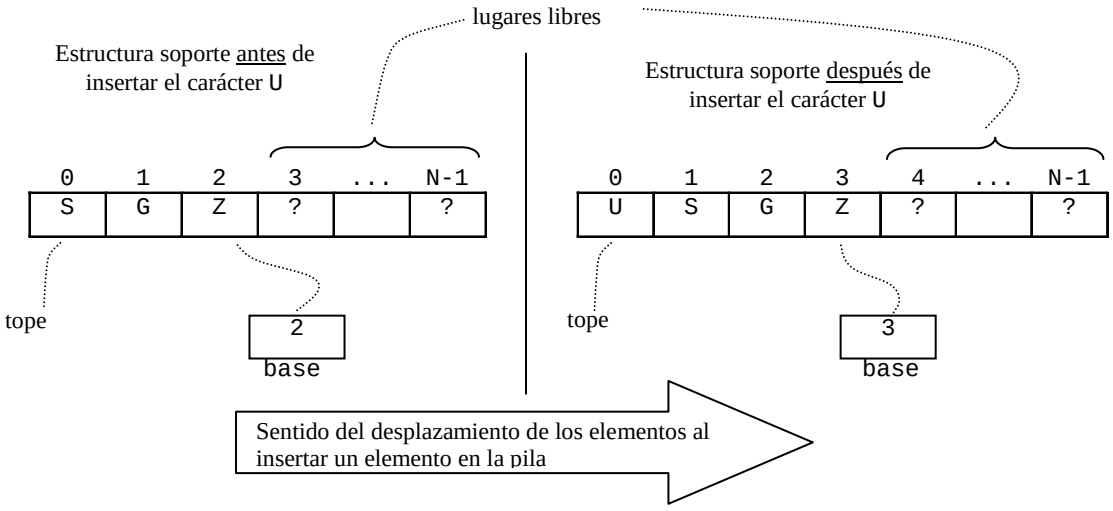


**Figura 11.2. Inserción en una pila de caracteres implementada con el método de desplazamiento, con variable para la base y el tope fijo en la primera posición del arreglo soporte.**

La Figura 11.2 muestra el estado de la estructura soporte (arreglo y variable para la base) antes y después de insertar en el tope el carácter '`U`'.

## CON UNA MARCA

En este caso, emplearemos una *marca* que sirva para saber dónde está, en el arreglo, la base, es decir el fin, de la

pila. Esta *marca* deberá ser un valor de los del tipo de los elementos de la pila que solo podrá ser utilizado para esto y no como un valor de la pila simulada, lo que obliga a reducir el conjunto de los valores posibles de los elementos en uno. Esta marca ocupa un lugar en el arreglo, lo que también reduce en uno la cantidad de espacio destinado a los elementos en la estructura soporte. La marca puede servir para el control del desfonde (está en la posición del tope, primera posición del arreglo); o para el desborde (está en la última posición del arreglo).

```
#define N     100
#define MARCA '*'

char miPila[N];
```

**Figura 11.3. Definición de la pila de caracteres `miPila` usando como estructura soporte un arreglo para los elementos. La constante simbólica MARCA es el valor distinguido empleado en la pila para indicar la base de la misma.**

La marca deberá desplazarse, junto con el resto de los elementos de la pila, cada vez que se haga una supresión o una inserción. Inicialmente, cuando la pila está vacía, la marca se encontrará en la posición 0 del arreglo, y se irá moviendo a medida que se insertan y se suprimen elementos. La Figura 11.4 muestra el estado de la estructura soporte (arreglo con marca) antes y después de suprimir en el tope el carácter 'S'.

Los espacios libres se administran automáticamente en forma dinámica. En efecto, puede decirse que están organizados como una pila cuyo tope "toca" la base de la pila simulada, de manera tal que cada vez que inserta un elemento en la pila simulada se produce automáticamente una supresión en la pila de los espacios libres y viceversa.

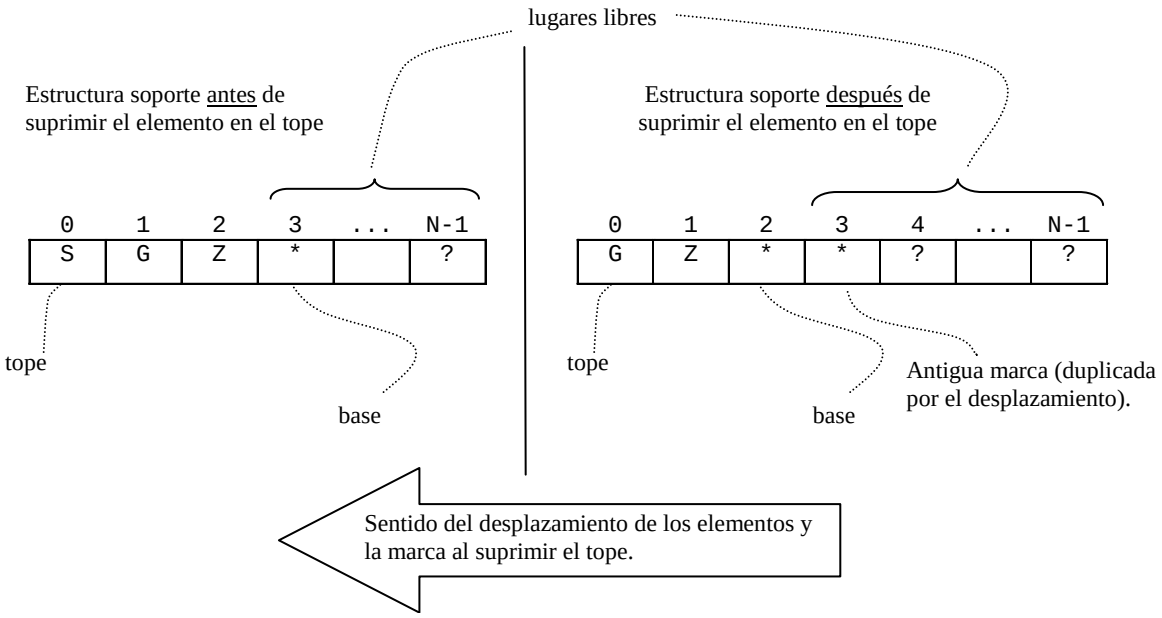


**Figura 11.4. Supresión en una pila de caracteres implementada con el método de desplazamiento, con marca para la base y el tope fijo en la primera posición del arreglo soporte.**

Algo similar ocurre si se fija el tope de la pila en la última posición del arreglo, en lugar de la primera como se había considerado hasta aquí. En este caso, los desplazamientos que se deberán hacer en el arreglo cada vez que ocurre una inserción o una supresión en la pila simulada tendrán un "sentido" inverso al "sentido" que tienen cuando el tope de la pila simulada se fija en el primer elemento del arreglo.

## SIN DESPLAZAMIENTO

En este caso, se fija la base de la pila, en la primera posición del arreglo y se mantiene una variable que indica en

qué posición del arreglo se encuentra el elemento del tope. En la Figura 11.5 se muestra una definición posible para esta estructura soporte. Se ha usado un registro para encapsular la estructura que consiste de un arreglo para almacenar los elementos de la pila (`elementos`) y un campo (`tope`) que mantiene el índice del arreglo en donde se encuentra el elemento del tope de la pila.

```
#define N    80

struct pila {
   char elementos[N];
   int tope;
} miPila;
```

**Figura 11.5. Definición del tipo `struct pila` usando como estructura soporte un arreglo para los elementos de la pila `miPila` y una variable `tope` para indicar el tope de la pila.**

La variable `tope`, se la incrementa en `1` cada vez que se hace una inserción (push) en la pila o se la decrementa en `1` cada vez que se hace una supresión (pop). También puede servir para controlar el desfonde (cuando es menor que `0`, ya que en C `0` es el mínimo valor del índice del arreglo) y para el desborde (cuando es mayor que `N-1`), ya que `N-1` es el máximo valor del índice del arreglo `elementos`.

Los espacios libres se administran automáticamente en forma dinámica. En efecto puede decirse que están organizados como una pila cuyo tope "toca" el tope de la pila simulada y sobre la que se hace una supresión automáticamente cada vez que se inserta en la pila simulada y viceversa. La variable que indica dónde está el tope de la pila simulada, permite calcular al mismo tiempo dónde está el tope de la pila de los espacios libres.

La Figura 11.6 muestra el estado de la estructura soporte (arreglo más variable para el tope) antes y después de insertar en el tope el carácter '`U`'.

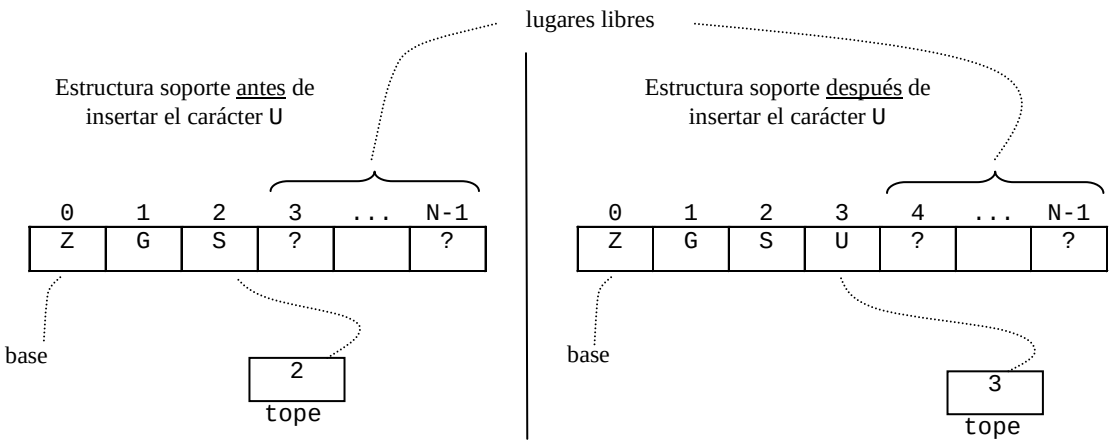


**Figura 11.6. Inserción en una pila de caracteres implementada con el método sin desplazamiento, con variable para el tope y la base fija en la primera posición del arreglo soporte.**

Como una posible alternativa para esta implementación, la base podría fijarse en la última posición del arreglo y hacer que el tope varíe, incrementándolo cada vez que se hace una supresión (pop) o decrementándolo cada vez que se hace una inserción (push).

## 11.4 IMPLEMENTACIÓN DE FILAS O COLAS

Una forma posible de simular filas es empleando un arreglo cuyo tipo base sea igual al de los elementos de la fila

que va a simularse.

También para esta estructura presentamos, a modo de ejemplo, tres métodos posibles de implementación o simulación, dos con desplazamiento y uno sin desplazamiento, todos usan como estructura soporte un arreglo. Estos métodos no son los únicos, existen variantes tanto con administración dinámica de los espacios libres como estática. Queda al alumno analizar otras variantes.

## CON DESPLAZAMIENTO

En este caso se fija el primero de la fila en la primera posición del arreglo y se hace un desplazamiento en el arreglo de los elementos de la fila cada vez que se hace una supresión. Para la inserción no será necesario realizar un desplazamiento, simplemente se insertará después del último de la fila.

En consecuencia, será necesario tener una forma de indicar dónde está, en el arreglo, el último elemento de la fila. Esto puede hacerse de dos maneras: usando una variable que mantiene la posición en el arreglo del último elemento de la fila o usando una marca que indica donde se acaba la fila.

### CON VARIABLE PARA EL ÚLTIMO

En este método, usamos una variable que sirve para saber dónde se encuentra, en el arreglo, el último elemento de la fila, es decir, el fin de la fila. Esta variable se deberá incrementar cada vez que se haga una inserción, dado que en las filas las inserciones son detrás del último, y se deberá decrementar cuando se haga el desplazamiento después de una supresión.

En la Figura 11.7 se muestra una definición posible para esta estructura soporte. Se ha usado un registro para encapsular la estructura que consiste de un arreglo para almacenar los elementos de la fila (`elementos`) más un campo (`ultimo`) que mantiene el índice del arreglo en donde se encuentra el `ultimo` elemento de la fila. Inicialmente, cuando la fila se encuentra vacía, el valor de `ultimo` debe ser un valor menor al primer índice del arreglo, el cual en el caso de esta definición debería ser `-1`.

```
#define SIZE  120

struct fila {
   char elementos[SIZE];
   int ultimo;
} miFila;
```

**Figura 11.7. Definición del tipo `struct fila` usando como estructura soporte un arreglo para los elementos de la fila `miFila` y una variable `ultimo` para indicar el último elemento de `miFila`.**

La variable `ultimo` puede servir para el control del desfonde de la fila cuando sea menor que `0`, ya que `0`, en C, es el mínimo valor del índice de un arreglo y para el desborde cuando sea mayor que `N-1`, siendo `N-1` el máximo valor del índice del arreglo.

La Figura 11.8 muestra el estado de la estructura soporte (arreglo y variable para el último) antes y después de insertar el caracter `U` después del último.

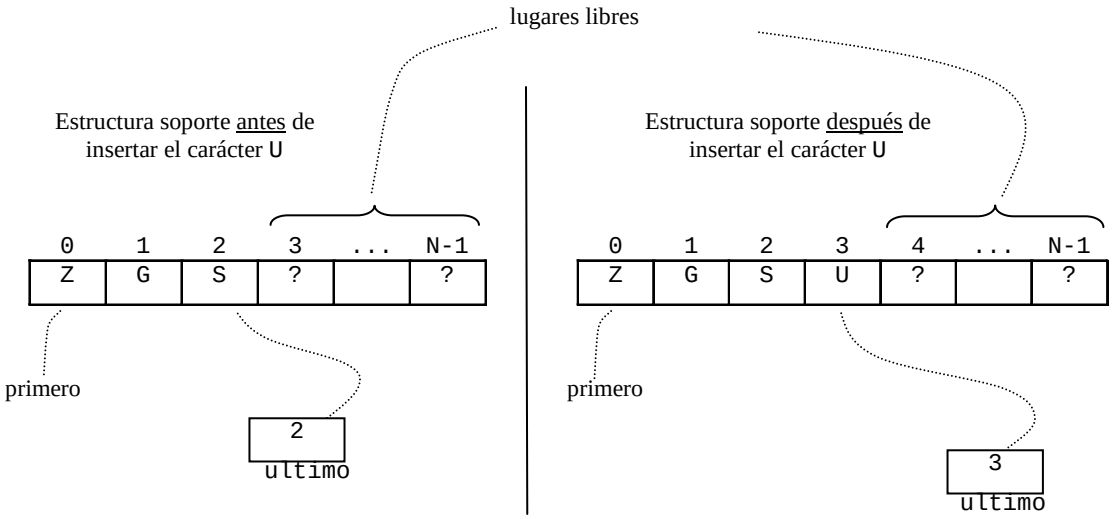


**Figura 11.8. Inserción en una fila de caracteres implementada con el método de desplazamiento, con variable para el último y el primero fijo en la primera posición del arreglo soporte.**

La Figura 11.9 muestra el estado de la fila antes y después de suprimir el primer elemento de la fila, en este ejemplo el carácter Z.

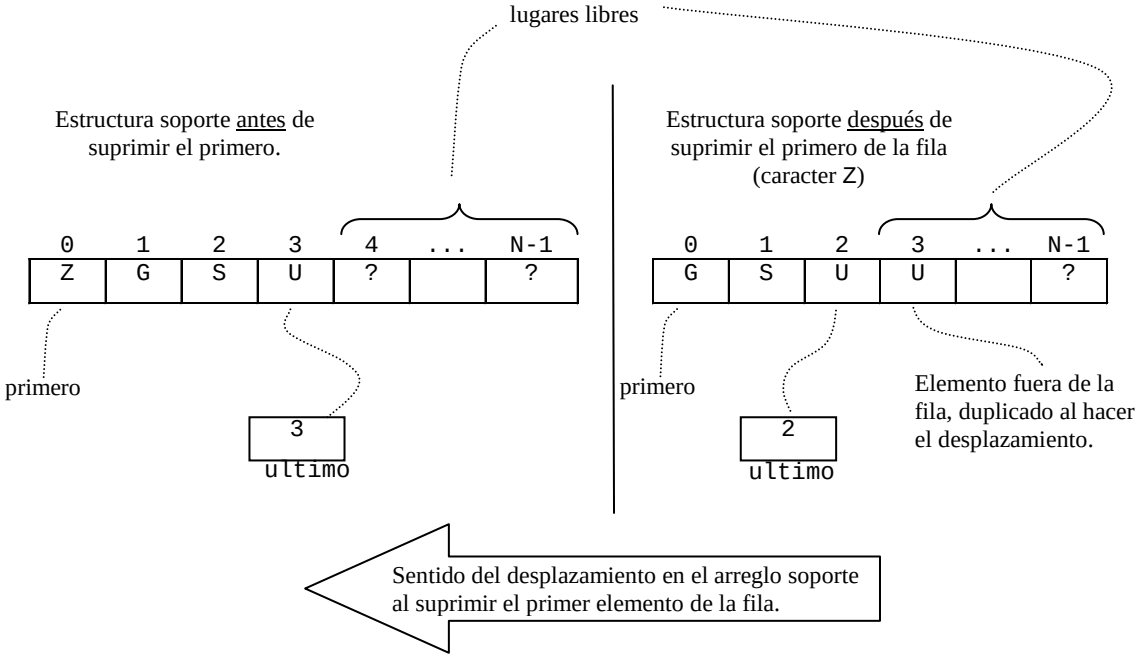


**Figura 11.9. Supresión en una fila de caracteres implementada con el método de desplazamiento, con variable para el último y el primero fijo en la primera posición del arreglo soporte.**

### CON UNA MARCA

Este método de implementación usa una *marca* que sirve para saber dónde está, en el arreglo, el último, (es decir el final de la fila). Esta marca deberá ser un valor de los del tipo de los elementos de la fila, que solo podrá ser utilizado para esto y no como VIPD, lo que obliga a reducir el conjunto de los valores posibles de los elementos en

uno.

La Figura 11.10 muestra una posible definición en C de una fila de caracteres de nombre `miFila` que usa como estructura soporte un arreglo de `100` caracteres. Asimismo, se muestra la definción de una constante simbólica `MARCA` usada en esta implementación como marca de final de la fila. Como marca se ha usado el carácter `'*'` pero podría haberse empleado cualquier otra carácter que no forme parte del conjunto de valores válidos de la VIPD.

```
#define N     100
#define MARCA '*'

char miFila[N];
```

**Figura 11.10. Definición de la fila de caracteres `miFila` usando como estructura soporte un arreglo para los elementos. La constante simbólica MARCA es el valor distinguido empleado en la fila para indicar el final de la misma.**

La marca ocupa un lugar en el arreglo lo que reduce en uno la cantidad de espacio destinado a los elementos en la estructura soporte. La marca puede servir para el control del desfonde (está en la posición del primero de la fila, posición `0` del arreglo, en este caso); o para el desborde (está en la última posición del arreglo, posición `N-1`, en este caso).

Esta marca deberá desplazarse, junto con el resto de los elementos de la fila, cada vez que se haga una supresión. En la Figura 11.11 se muestra el estado de la estructura soporte antes y después de suprimir el primer elemento de la fila. Todos los elementos fueron desplazados un lugar a la izquierda, incluida la marca.

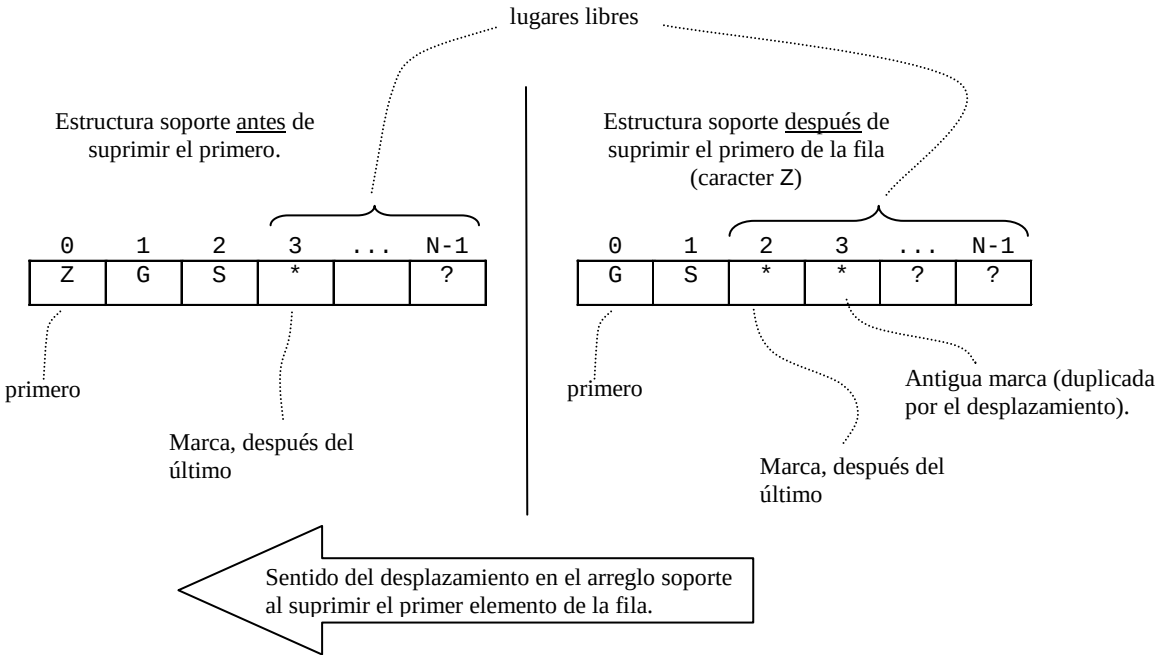


**Figura 11.11. Supresión en una fila de caracteres implementada con el método de desplazamiento, con marca para la base y el primero fijo en la primera posición del arreglo soporte.**

Para el caso de las inserciones, la marca deberá moverse a la posición siguiente del arreglo de la que se encuentra para liberar el espacio necesario para la inserción del nuevo elemento. La Figura 11.12 muestra el estado de la estructura soporte antes y después de insertar un elemento en la fila.

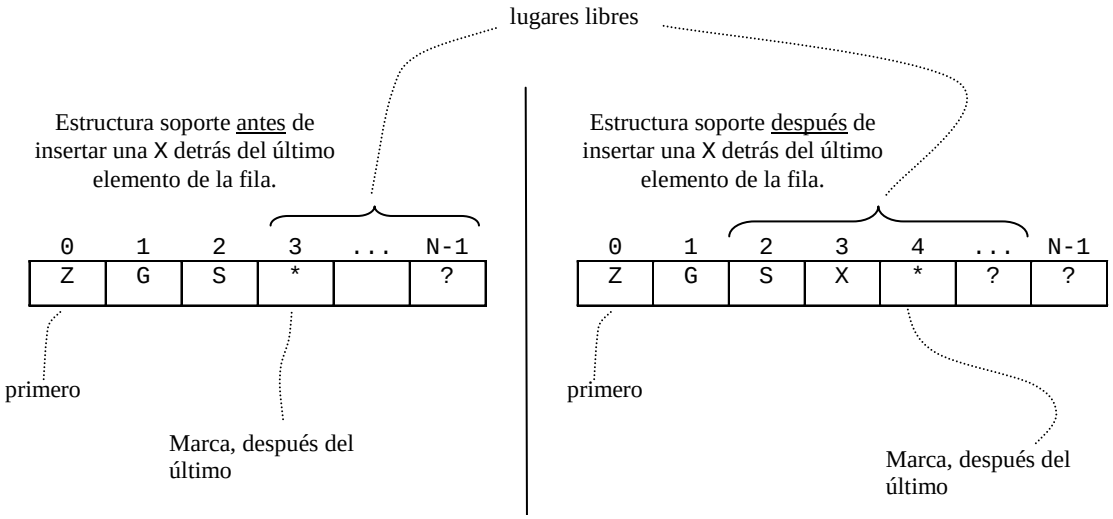


**Figura 11.12. Supresión en una fila de caracteres implementada con el método de desplazamiento, con marca para la base y el primero fijo en la primera posición del arreglo soporte.**

Como una alternativa a estos dos métodos, algo similar ocurre si se fija el primero de la fila en la última posición del arreglo:

En el caso del método con marca, será necesario hacer un desplazamiento hacia la derecha de los elementos de la fila, que están en el arreglo, incluida la marca, cada vez que se haga una supresión, y se deberá mover al lugar anterior la marca, cada vez que se haga una inserción, para cederle esa posición al nuevo último elemento.

En el caso del método con variable para el último, la variable que sirva de puntero para saber dónde está, en el arreglo, el último, (es decir el fin de la fila) se deberá decrementar cada vez que se hace una inserción y deberá incrementar cuando se haga una supresión.

Otra alternativa para el método con la marca: en lugar de fijar el primer elemento de la fila en alguno de los extremos del arreglo que sirve como soporte, como se explicó en los párrafos anteriores, puede fijarse el último en alguno de los extremos y tener un puntero o una "marca" para el primero. En este caso el desplazamiento solo se hará cuando se haga una inserción.

El manejo de los espacios libres es dinámico, y los mismos se encuentran todos juntos y contiguos después del primero (si se fijó el último), o después del último (si se fijó el primero). Puede decirse que los espacios libres se encuentran organizados como una pila en la que se inserta (push) cada vez que se suprime un elemento de la fila y se suprime (pop) cada vez que se inserta un elemento en la fila.

## SIN DESPLAZAMIENTO

En este método el arreglo se maneja en *forma circular*. Se emplean dos variables que mantienen las posiciones del primer elemento y del último de la fila. Cada vez que se inserta un nuevo elemento, se incrementa en uno la variable que señala al último y cada vez que se suprime se incrementa en uno la variable que señala al primero de la fila.

En la Figura 11.13 puede verse una definición posible en C de una estructura soporte para una fila de caracteres. El arreglo de caracteres `elementos`, que se administra en forma circular, mantendrá los elementos de la fila, mientras que las variables `p` y `u` servirán para mantener los índices del arreglo que indican dónde se encuentran el primer y el último elemento de la fila, respectivamente. La variable `c`, que no es estrictamente necesaria pero cuyo

uso simplifica la programación, es usada para conocer la cantidad de elementos que hay en la estructura soportada en todo momento.

```
#define N   8

struct fila {
     char elementos[N];
     int p; /*posición del primer elemento*/
     int u; /*posición del ultimo elemento*/
     int c; /*cantidad de elementos*/
} miFila;
```

**Figura 11.13. Definición de la fila de caracteres `miFila` usando como estructura soporte un arreglo para los elementos y dos cursores que indican la posición del primer y ultimo, elemento más una variable `c` que mantiene la cantidad de elementos en la fila.**

Dado que las operaciones de inserción y supresión pueden hacer que los valores de `p` o `q` o ambos, en algún momento, lleguen a ser mayores que la última posición del arreglo, la operación de suma debe hacerse de manera tal que llegado a ese caso se vuelva a tener el valor que indique al primer elemento del arreglo. Esto equivale a decir que la operación de incremento se hace módulo el tamaño del arreglo y puede hacerse con una sentencia `if` o con el operador de módulo (`%` en C).

Por supuesto que, en todos los casos, deberán tenerse en cuenta las condiciones de desborde y desfonde (overflow y underflow). Estas condiciones pueden controlarse de diversas maneras: con una variable que mantenga la cantidad de elementos presentes en la fila simulada, que en la estructura de la Figura 11.13 corresponde al campo `c`, o por medio de los punteros que apuntan al primero y al último. En éste último caso, la condición debe ser cuidadosamente estudiada y todos los casos posibles analizados.

Como alternativa a esta implementación, en lugar de incrementar los punteros puede decrementárselos, manteniendo los mismos recaudos que cuando se incrementan.

En la Figura 11.14 se muestra gráficamente un ejemplo de una fila. Suponemos que se trata de una fila de caracteres como la que se encuentra dibujada en la Figura 11.14(a), que tiene tres elementos: `S`, `G`, `Z`. Esta fila podría implementarse en una variable de tipo `struct fila` como la del ejemplo `miFila` definida en la Figura 11.13.

Suponemos que el estado de esta variable `miFila`, luego de realizarse varias inserciones y supresiones, es el que se muestra en la Figura 11.14(b). Si se inserta el caracter `T` en la fila ésta quedará como se puede ver en la Figura 11.14(c). La estructura que soporta esta fila quedaría, entonces, como se ve en la Figura 11.14(d).

Si ahora insertamos `H`, la cola queda como se ve en la Figura 11.15(a), y la estructura soporte como se muestra en la Figura 11.15(b). Si luego se hace una supresión del primer elemento, la cola quedará como la de la Figura 11.15(c), y la estructura soporte como la de la Figura 11.15(d).

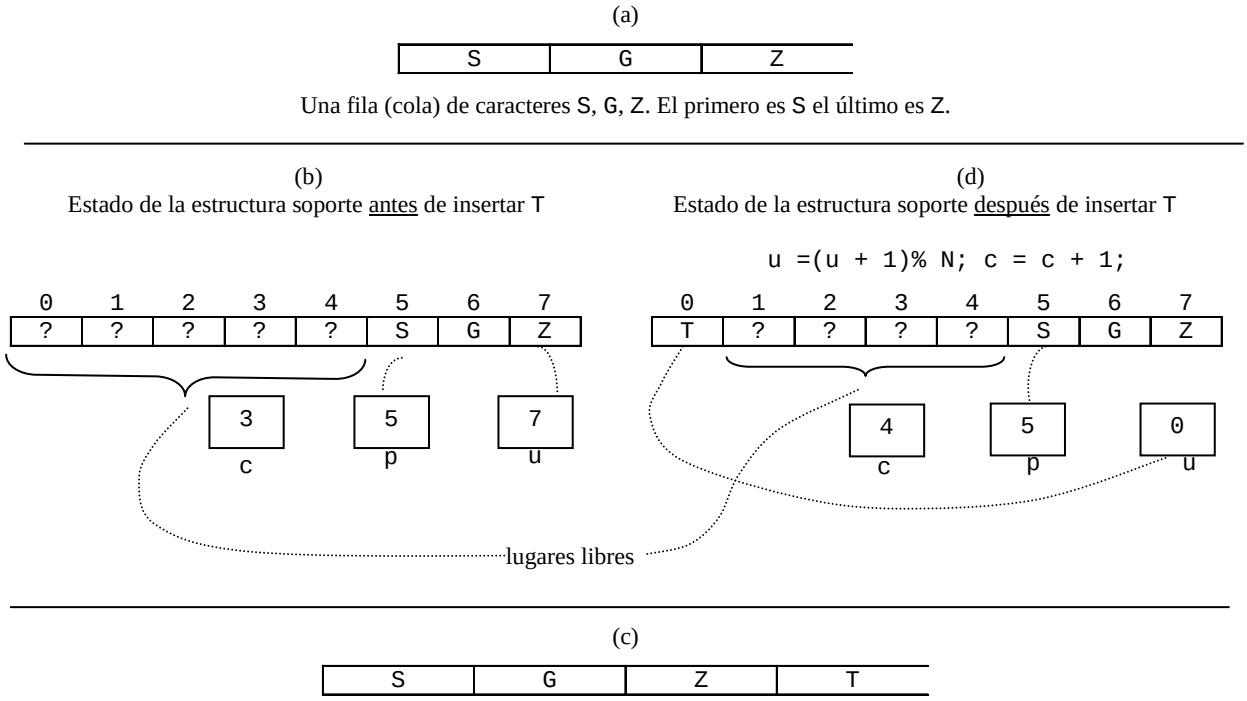


**Figura 11.14. Ejemplo de inserción en una cola de caracteres, implementada manejando el arreglo en forma circular.**

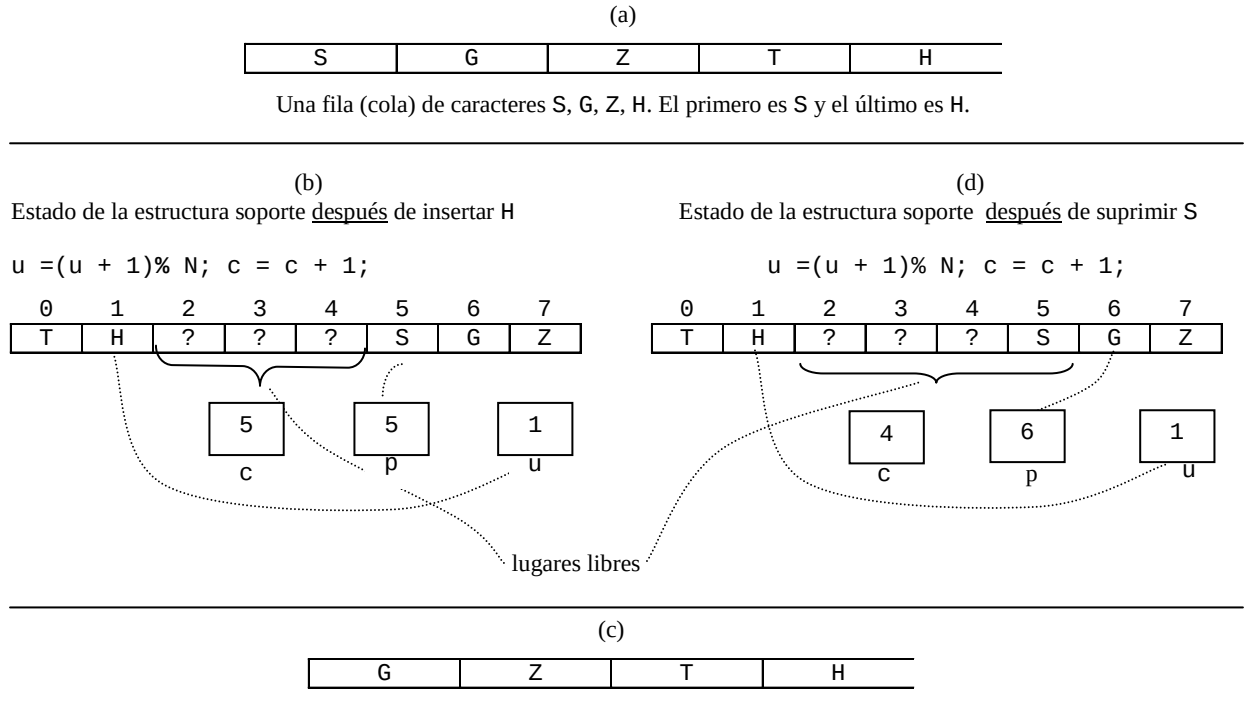


**Figura 11.15. Ejemplo de inserción y supresión en una cola de caracteres, implementada manejando el arreglo en forma circular.**

En este método, el manejo de los espacios libres es dinámico y automático, y los mismos se encuentran todos juntos y contiguos entre el primero y el último de la fila simulada. Puede decirse que los espacios libres se encuentran organizados, a su vez, como una fila en la que se inserta cada vez que se suprime un elemento de la fila simulada y se suprime cada vez que se inserta un elemento en la fila simulada.

Estas dos filas (la simulada y la de los espacios libres) son como dos serpientes que se "muerden la cola", si se imagina que las bocas de las serpientes son el último de cada fila y las colas de las serpientes el primero de cada fila. Cuando la boca de una se come un pedazo de la cola de la otra hay una inserción y una supresión, respectivamente, en cada una de las filas.

Cabe notar que si no se hace un manejo circular del arreglo, entonces la administración de los espacios libres será estática, ya que se podrán hacer inserciones en la fila simulada usando los espacios libres mientras el puntero al último elemento no exceda la capacidad máxima del arreglo. En este caso, podrá optarse por realizar un garbage collection (estática temporaria) para reorganizar los espacios libres, o no hacer nada (estática permanente).

## 11.5 IMPLEMENTACIÓN DE LISTAS

Las listas son estructuras muchos más generales que las pilas y las colas ya que permiten que las operaciones de inserción, supresión e inspección de sus elementos sean hechas en cualquier posición de la misma, siempre que esa posición corresponda al *elemento corriente* (el elemento apuntado por el *cursor* asociado a la lista).

Dada su mayor generalidad y flexibilidad operacional, pueden servir más fácilmente para simular otras estructuras. Algunos autores sostienen, por ejemplo, que el arreglo es un caso particular de una lista: una lista contigua, es decir sin punteros, ya que estos no serían necesarios puesto que el siguiente elemento siempre está al lado del anterior.

En esta sección presentamos, a modo de ejemplo, tres métodos de implementación de listas que hacen uso de un arreglo como estructura soporte. Dos de los métodos son con deplazamiento de los elementos y uno es sin desplazamiento. En el capítulo 12, se presenta otro método que hace manejo dinámico de la memoria. Cabe aclarar que estos métodos no son los únicos. Queda al alumno analizar otras variantes.

### CON DESPLAZAMIENTO

En el método con desplazamiento empleamos un arreglo, cuyo tipo base debe ser igual al de las VIPDs de la lista que va a simularse, más una variable para el cursor de la lista.

En este caso, se fija el primero de la lista en la posición uno del arreglo y se hace un desplazamiento de los elementos de la lista, que están en el arreglo, cada vez que se hace una supresión o una inserción.

Será necesario tener una forma de indicar dónde está, no sólo el cursor en el arreglo sino también, el último elemento de la lista. Esto puede hacerse de dos maneras: usando una variable que mantiene la posición en el arreglo del último elemento de la lista o usando una marca para indicar dónde termina la lista.

#### CON VARIABLE PARA EL ÚLTIMO

En este método, usamos una variable que sirva para saber dónde está, en el arreglo soporte, el último elemento y así poder reconocer el fin de la lista. Esta variable se deberá incrementar en uno cada vez que se haga una inserción, después de hacer un desplazamiento a derecha desde el último elemento hasta la posición del cursor (en donde se insertará el nuevo elemento); y deberá decrementarse en uno, después de una supresión, cuando se haga un desplazamiento a izquierda desde el cursor hasta el último elemento. Esta variable puede servir para el control del desfonde cuando sea menor que `0` (si es `0` quiere decir que hay un elemento, el último ý único elemento, que se encuentra en la posición `0` del arreglo soporte); y para el desborde cuando sea mayor que `N-1`.

En la Figura 11.16 puede verse una posible definición en C de una estructura soporte para una lista de caracteres usando una variable para indicar el último elemento de la lista. El arreglo de caracteres `vipd` mantendrá los elementos de la lista, mientras que las variables `cur` y `ultimo` servirán para mantener los valores de los índices del arreglo dónde se encuentran el cursor de la lista y el último elemento de la misma, respectivamente.

```
#define N  100

struct lista{
   char vipd[N]; /* para las vipds de la lista */
   int cur;      /* cursor de la lista */
   int ultimo;   /* posición del ultimo elemento de la lista */
} miLista;
```

**Figura 11.16. Definición de la lista de caracteres `miLista` usando como estructura soporte un arreglo para los elementos, una variable para el cursor y otra para el último elemento.**

La Figura 11.17 muestra el estado de la estructura soporte para una lista de caracteres antes y después de realizar la inserción del carácter `U` en la posición apuntada por el cursor. El cursor, luego de la inserción, deberá quedar en la misma posición, mientras que el valor de la variable `ultimo` es incrementada en `1`.

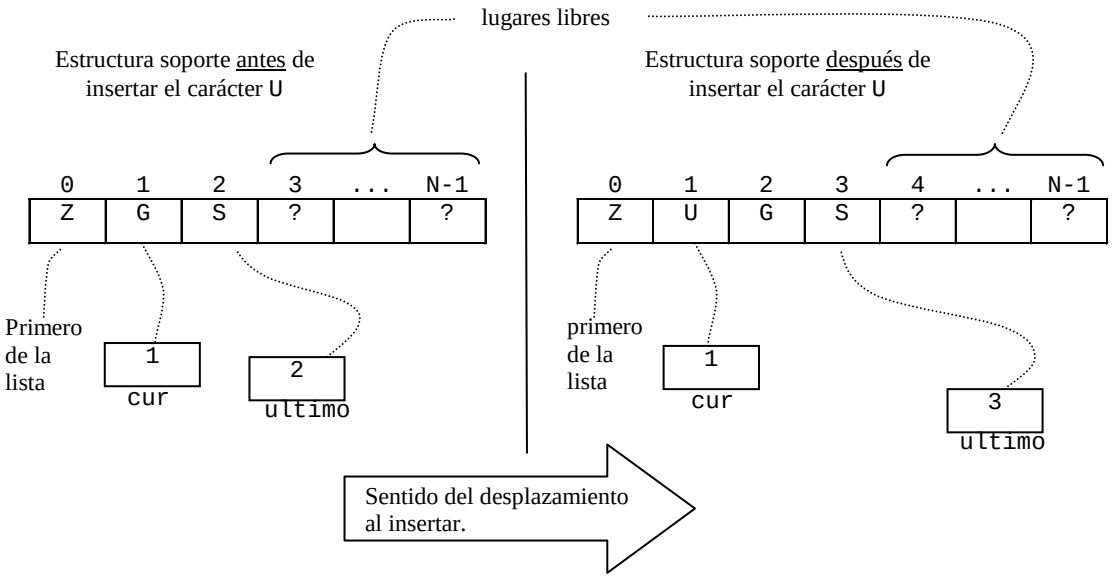


**Figura 11.17. Inserción en una lista de caracteres en el lugar apuntado por el cursor, implementada con el método de desplazamiento, con variable para el último y el primero de la lista fijo en la primera posición del arreglo soporte.**

La Figura 11.18 muestra el estado de la estructura soporte para una lista de caracteres antes y después de realizar la supresión del carácter apuntado por el cursor. El cursor, luego de la inserción, deberá quedar en la misma posición, mientras que el valor de la variable `ultimo` es decrementado en `1`.

En este caso, la administración de los espacios libres es dinámica y automática. Los espacios libres están organizados como una pila cuyo tope 'toca' el último de la lista. La variable que señala al último elemento de la lista permite saber dónde está el tope de la pila de los espacios libres.

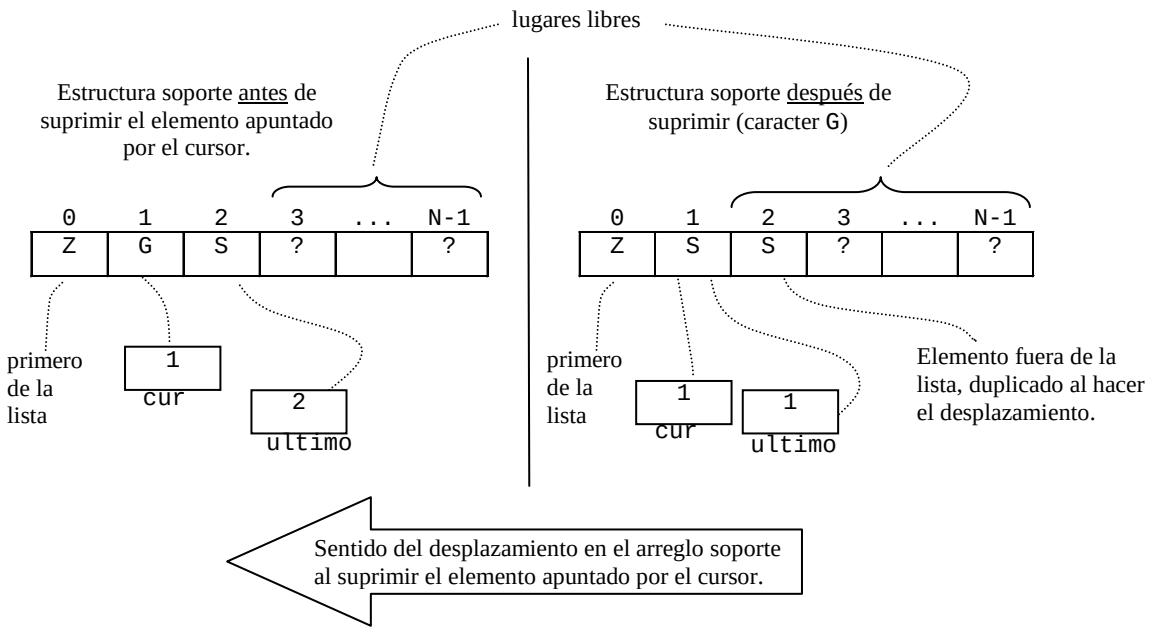


**Figura 11.18. Supresión del elemento apuntado por el cursor de una lista de caracteres implementada con el método de desplazamiento, con variable para el último y el primero fijo en la primera posición del arreglo soporte.**

## CON UNA MARCA

Este método usa, además de un arreglo para mantener los elementos de la lista, una *marca* que sirve para saber en todo momento dónde está, en el arreglo, el último elemento y así conocer dónde está el fin de la lista. Esta marca deberá ser un valor de los del tipo de las VIPDs de la lista, que quedará reservado para este fin, sin poder ser utilizado como VIPD. Esto obliga a reducir el conjunto de los valores posibles de las VIPDs en uno.

Asimismo, la marca ocupa un lugar en el arreglo, lo que reduce en uno la cantidad de espacio destinado a las VIPDs en el arreglo soporte. La marca puede servir para el control del desfonde (está en la primera posición del arreglo); o para el desborde (está en la última posición del arreglo). Además, deberá desplazarse, junto con el resto de los elementos de la lista, cada vez que se haga una supresión o una inserción.

En la Figura 11.19 puede verse una posible definición en C de una estructura soporte para una lista de caracteres usando un arreglo y una variable para el cursor. El arreglo de caracteres `vipd` mantendrá los caracteres de la lista, mientras que la variable `cur` servirá para mantener el valor del índice del arreglo que indica dónde se encuentra el cursor de la lista, es decir, indicar cuál es el elemento corriente sobre el cual se puede operar.

```
#define   MARCA   '*'
#define   N       100

struct lista{
   char vipd[N]; /* para las vipds de la lista */
   int cur;      /* cursor de la lista */
} miLista;
```

**Figura 11.19. Definición en C de la lista de caracteres `miLista` usando como estructura soporte un arreglo para los elementos, una variable para el cursor y una marca para indicar el final del la misma.**

Las Figura 11.20 muestra el estado de la estructura soporte antes y después de suprimir el elemento apuntado por

`cur` (la letra `G`, en la segunda posición de la lista). Notar que `cur` queda después de la supresión en el mismo lugar que se encontraba antes de la operación.

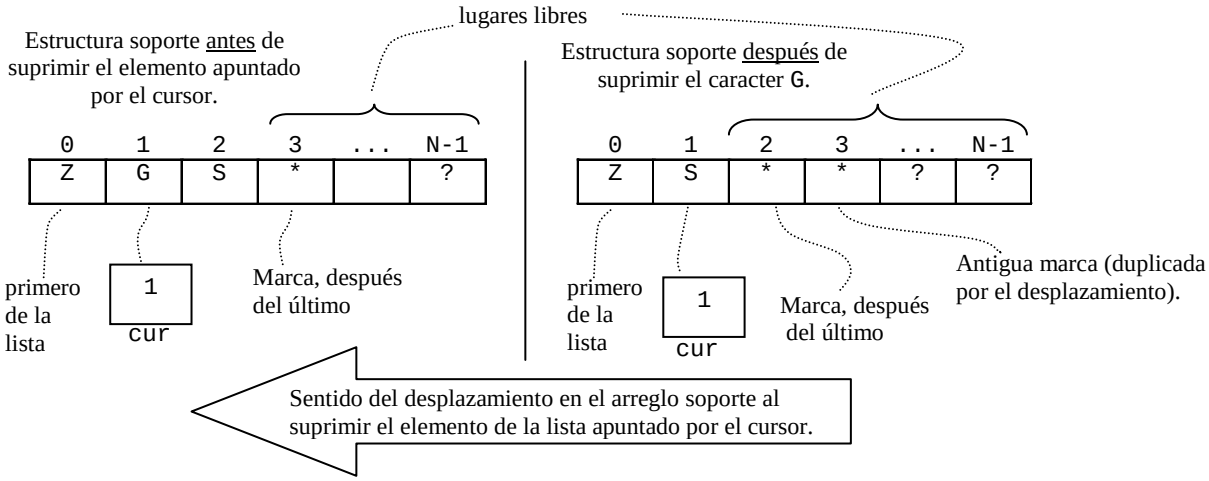


**Figura 11.20. Supresión en una lista de caracteres implementada con el método de desplazamiento, con marca para indicar el final de la lista y el primero fijo en la primera posición del arreglo soporte.**

La Figura 11.21 muestran el estado de la estructura soporte antes y después de insertar el carácter `M` en la posición apuntada por `cur`. Notar que `cur` queda en el mismo lugar que se encontraba antes de la inserción.

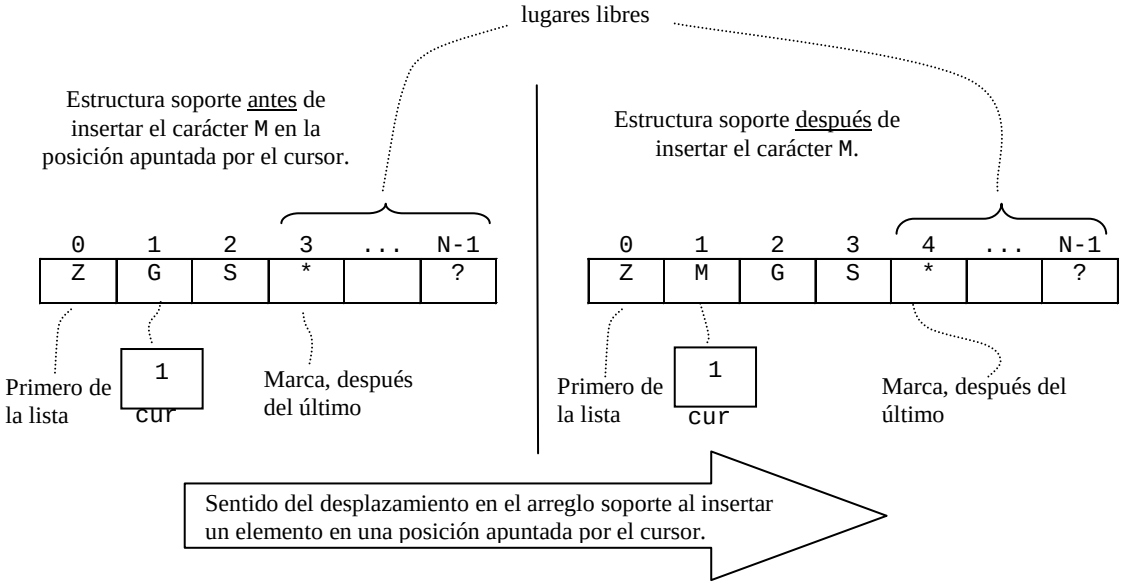


**Figura 11.21. Inserción en una lista de caracteres implementada con el método de desplazamiento, con marca para indicar el final de la lista y el primero fijo en la primera posición del arreglo soporte.**

Como una alternativa a estos dos métodos con desplazamiento, algo similar ocurre si se fija el primero de la lista en la última posición del arreglo. En este caso el desplazamiento de los elementos de la lista, que están en el arreglo, cada vez que se hace una supresión o una inserción tendrán un "sentido" opuesto.

Además, en el caso del método con variable para el último, la variable que sirva de puntero para saber dónde está, en el arreglo, el último, (es decir el fin de la lista) se deberá decrementar cada vez que se hace una inserción y deberá incrementar cuando se haga una supresión.

Otra alternativa, en el caso del método con marca, en lugar de fijar el primer elemento de la lista en alguno de los extremos del arreglo que sirve como soporte, como se explicó en los párrafos anteriores, es fijar el último en alguno de los extremos y tener un puntero o una "marca" para el primero.

El desplazamiento solo se hará, para todos los elementos de la lista, cuando se trata de una inserción o supresión al primero o al último (dependiendo de cual se haya fijado); y parcialmente si la inserción o supresión se hace en algún elemento intermedio. Es decir que se desplazan sólo aquellos elementos que cubren el lugar del suprimido o que liberan un lugar para insertar uno nuevo.

El desplazamiento en estos casos se hará desde el lugar que se quiere liberar (caso de la inserción), o hacia el lugar que se quiere ocupar (caso de la supresión), hasta el primero (si se fijó el último) o el último (si se fijó el primero).

La administración de los espacios libres es dinámica y automática, y los mismos se encuentran todos juntos y contiguos después del primero (si se fijó el último), o después del último (si se fijó el primero). Puede decirse que los espacios libres se encuentran organizados como una pila en la que se inserta (push) cada vez que se suprime un elemento de la lista y se suprime (pop) cada vez que se inserta un elemento en la lista.

### SIN DESPLAZAMIENTO

En este método de implementación de listas, cada elemento de la lista a ser simulada debe constituirse con la VIPD y el o los punteros (para las bidireccionales habrá dos).

### CON ARREGLOS SEPARADOS

Se emplearan tantos arreglos como sean necesarios. Por ejemplo, para simular una lista unidireccional será necesario contar con dos arreglos: uno para las VIPDs de la lista, cuyo tipo base será el tipo de los elementos de la lista. Otro para los punteros, cuyo tipo base será uno que sirva para indexar el arreglo reservado para las VIPDs. En este tipo base, deberemos reservar un valor distinguido para representar a *nil*, el cual no podrá ser uno de los valores empleados para direccionar el arreglo. Además, se necesitará una variable que sirva para el acceso a la lista, es decir, servirá para apuntar al lugar en el arreglo donde se encuentre el primer elemento de la lista.

Veamos un ejemplo de definición de un tipo en C de una estructura que sirva como soporte de una lista bidireccional de caracteres.

```
#define N   9;

struct blista{
     char vipd[N];    /* para las vipds de la lista */
     int ps[N];       /* para los punteros al siguiente elemento */
     int pa[N];       /* para los punteros al elemento anterior */
     int acc;         /* acceso a la lista */
     int lib;         /* acceso a lista de espacios libres */
     int cur;         /* cursor de la lista */
     int curaux;      /* cursor auxiliar */
} miLista;
```

**Figura 11.22. Una definición en C de la lista bidireccional de caracteres `miLista` usando como estructura soporte un arreglo para las VIPDs y otros dos para los punteros al siguiente y al anterior, un puntero al primero de la lista (`acc`), un puntero al primero de la lista de espacios libres (`lib`), un cursor (`cur`) y un cursor auxiliar (`curaux`).**

En esta estructura con tres arreglos separados, la posición i-ésima del arreglo con las VIPDs de la lista junto con la i-ésima de los arreglos de los punteros constituirán un elemento de la lista simulada con sus tres componentes: VIPD, puntero al siguiente y puntero al anterior (si es bidireccional, por supuesto).

Cada elemento del arreglo de VIPDs junto con el correspondiente elemento en cada uno de los arreglos de punteros constituye un elemento de la lista soportada. Por lo tanto, la composición de la posición i-esima de cada uno de los arreglos corresponderá a un único elemento en la lista. Notar que ese elemento no necesariamente se encontrará en la i-esíma posición de la lista. Por otro lado, los valores almacenados en cada elemento de los arreglos de punteros (siguiente y anterior) deberán tener valores que permitan conocer dónde se encuentran, en los tres arreglos, los elementos correspondientes de la lista soportada.

La variable que apunte al primer lugar de la lista, es decir el acceso a la lista, tendrá el valor de la posición en los arreglos (que es la misma en todos ellos) donde se encuentre el primer elemento de la lista simulada, el cual no es necesariamente el primer elemento en los arreglos.

La variable que sirve como *cursor auxiliar* tiene el propósito de apuntar al elemento anterior al apuntado por el *cursor*. Esto permite trabajar sin problemas, tanto en las supresiones como en las inserciones, para "enganchar" los punteros. Esto puede hacerse fácilmente en las listas bidireccionales (porque existe un puntero atrás) pero que resulta más difícil en las unidireccionales. De no existir este cursor auxiliar, para reacomodar los punteros, será necesario recorrer la lista (en las unidireccionales, por lo menos), desde el principio, para encontrar al elemento anterior al que apunta el cursor (que es el que se quiere eliminar o donde se quiere insertar).

Los espacios libres pueden ser administrados de las dos maneras mencionadas en la sección 11.2 "Administración de los espacios libres", es decir estática o dinámica.

En ambos casos será necesario tenerlos organizados en alguna estructura que permita emplearlos cada vez que se quiere insertar en la estructura simulada y, para el caso del manejo dinámico, dicha estructura (de los espacios libres) tendrá que permitir insertar cada vez que se suprima en la lista simulada.

Cuando el manejo sea dinámico, a los espacios libres conviene organizarlos también como una lista (es decir con punteros), ya que no necesariamente, luego de varias inserciones y supresiones, estarán contiguos en el arreglo. Esta lista de espacios libres, sin embargo, puede comportarse como una pila ya que las respectivas inserciones (cuando se suprime en la lista simulada) y supresiones (cuando se inserta en la lista simulada) pueden hacerse en el primer lugar de la misma.

Además, se necesitará una variable que apunte al lugar en el arreglo donde se encuentre el primer elemento de la estructura de los espacios libres (acceso a la lista de espacios libres). Esa variable recibe el nombre `lib` en la estructura del ejemplo (Figura 11.22).

En la Figura 11.23 se muestra una lista bidireccional. Para cada elemento de la bilista aparecen los valores correspondientes, tanto para las VIPDs como para los punteros, suponiendo una implementación empleando varios arreglos, como la dada en la Figura 11.24. El estado de la lista, así como el de los arreglos, no es necesariamente el inicial sino uno, de los tantos posibles, luego que se ha operado haciendo varias inserciones y supresiones en la lista. Notar que el valor `-1` fue elegido para representar a *nil*.

En la Figura 11.24 se muestra el estado de la estructura soporte con los arreglos separados, antes y después de una supresión. El antes corresponde a la lista de la Figura 11.23. Después de la supresión, la lista queda como se muestra en la Figura 11.25.

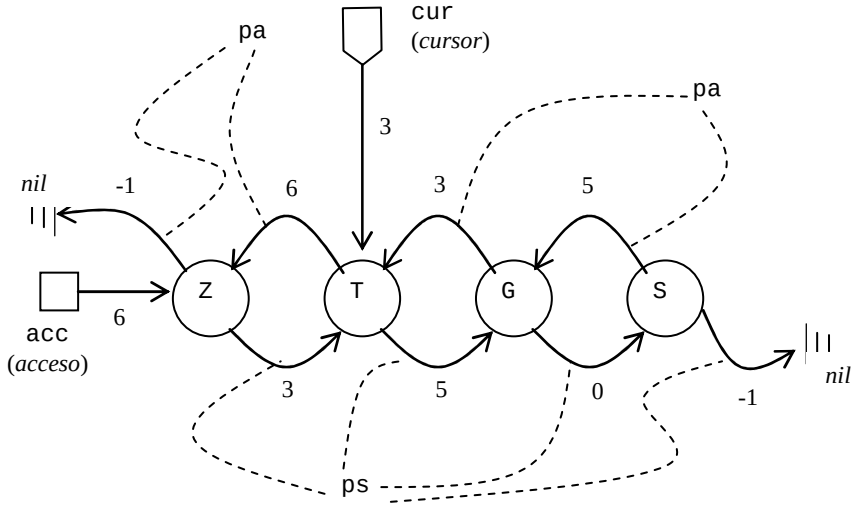


**Figura 11.23. Ejemplo de una lista bidireccional de caracteres con los valores de los campos de punteros `ps` (siguiente), `pa` (anterior), `acc` (acceso) y `cur` (cursor) de la estructura soporte de la Figura 11.24.**

Estado de la estructura soporte

antes de suprimir en la posición apuntada por `cur`

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| vipd | S | ? | ? | T | ? | G | Z | ? | ? |
| ps | -1 | 7 | 1 | 5 | 2 | 0 | 3 | 8 | -1 |
| pa | 5 | ? | ? | 6 | ? | 3 | -1 | ? | ? |

{ acc = 6, cur = 3, lib = 4 }

después de suprimir en donde indicaba `cur`

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| vipd | S | ? | ? | T | ? | G | Z | ? | ? |
| ps | -1 | 7 | 1 | 4 | 2 | 0 | 5 | 8 | -1 |
| pa | 5 | ? | ? | 6 | ? | 6 | -1 | ? | ? |

{ acc = 6, cur = 5, lib = 3 }

**Nota**: los lugares con ? no interesa qué valor tengan.

**Figura 11.24. Estado de la estructura soporte de la lista bidireccional de la Figura 11.23 administrada con arreglos separados**.

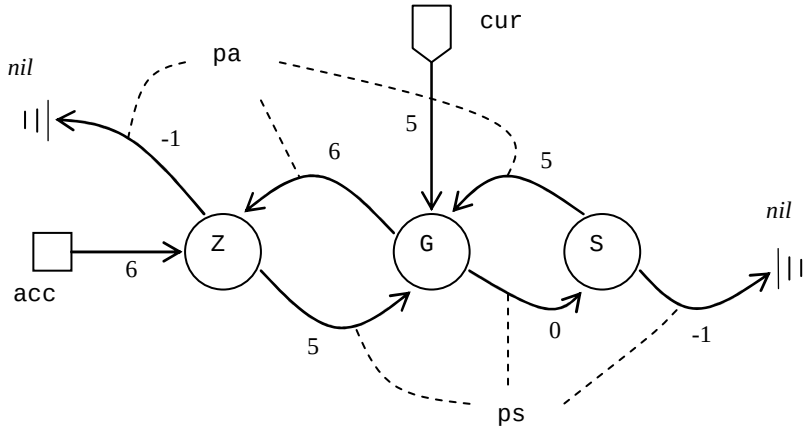


**Figura 11.25. Estado de la lista de la Figura 11.23 después de la supresión del elemento (T) apuntado por el cursor.**

La Figura 11.26 muestra el estado del arreglo antes (tal como había quedado la lista, en la Figura 11.25) y después de la inserción de la letra U.

Estado de los arreglos

antes de insertar el carácter U

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| vipd | S | ? | ? | T | ? | G | Z | ? | ? |
| ps | -1 | 7 | 1 | 4 | 2 | 0 | 5 | 8 | -1 |
| pa | 5 | ? | ? | 6 | ? | 6 | -1 | ? | ? |

{ acc = 6, cur = 5, lib = 3 }

después de insertar U en donde indicaba cur

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| vipd | S | ? | ? | U | ? | G | Z | ? | ? |
| ps | -1 | 7 | 1 | 5 | 2 | 0 | 3 | 8 | -1 |
| pa | 5 | ? | ? | 6 | ? | 3 | -1 | ? | ? |

{ acc = 6, cur = 3, lib = 4 }

**Nota**: los lugares con ? no interesa que valor tengan.

**Figura 11.26. Estado de la estructura soporte antes y después de insertar la letra U.**

En la Figura 11. 27 se ve cómo queda la lista luego de la inserción mostrada en la Figura 11.26.

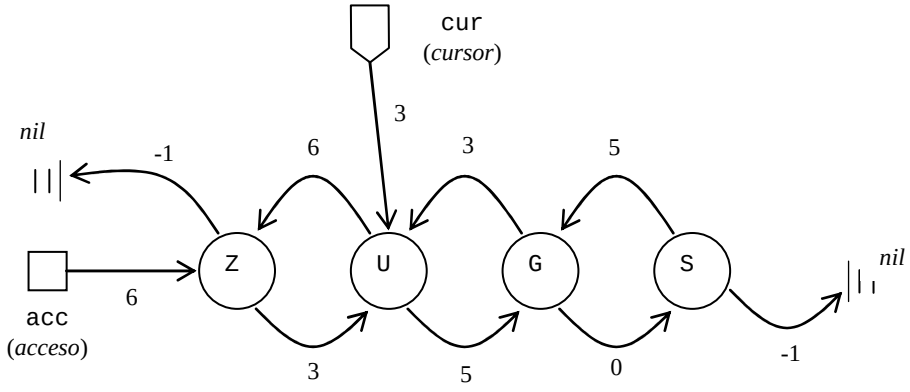


**Figura 11. 27. Estado de la lista de la Figura 11.25 después de la inserción de la letra U en la posición apuntada por el cursor.**

¿Cómo serían los procedimientos y funciones que implementan las operaciones posibles sobre una lista definida con la estructura soporte de la Figura 11.22?

La condición de *lista vacía* es sencilla: si el acceso es igual a `-1`, que equivale a *nil* en nuestra estructura soporte, entonces no hay nada en la lista.

La condición de *lista llena*, es decir, de cuándo no hay más lugar en la estructura soporte para seguir insertando, es también sencilla: si la estructura de los espacios libres está vacía no hay lugar para insertar. ¿Cuándo está vacía ésta estructura? Cuando el acceso a ella (`lib`) es igual a `-1`, que equivale a *nil* en nuestra estructura soporte.

¿Cuándo el cursor está *fuera de la estructura*? Esto es cuando el cursor es igual a `-1`, que equivale a *nil*.

¿Cómo se hace *avanzar* el cursor? Esto se logra al asignar al cursor el valor del puntero al siguiente del elemento apuntado por el cursor. Es decir hacer `miLista.cur = miLista.ps[miLista.cur]`. ¿Por qué? Porque hay que hacer que `cur` tome el valor del puntero al siguiente (`ps`), para que apunte justamente al siguiente elemento. Este valor se encuentra en `miLista` en el arreglo `ps`, en la posición apuntada en `miLista` por `cur`.

Para *insertar* hay que conseguir un lugar libre. Esto se hace usando el acceso a la estructura de los espacios libres. Como el nuevo elemento se inserta entre dos elementos existentes o en el primer lugar (que equivale a insertarlo "entre" el acceso y el primer elemento) o después del último, hay que controlar de qué caso se trata y si alguno de los casos no se reduce a alguno de los otros.

Por ejemplo, si es entre dos elementos habrá que hacer que el elemento anterior (el que es apuntado por el cursor auxiliar) apunte al nuevo y el nuevo apunte al corriente (que es apuntado por el cursor), y luego hacer que el cursor apunte al nuevo.

Para *suprimir* hay que hacer que el puntero al siguiente (`ps`), del elemento anterior al apuntado por el cursor (que está siendo apuntando por el cursor auxiliar `curaux`), apunte al elemento apuntado por el puntero al siguiente (`ps`) del elemento apuntado por el cursor (`cur`). Luego, hacer que el cursor (`cur`) apunte al elemento apuntado por el puntero al siguiente (`ps`) del elemento al que `cur` apunta (esto se parece bastante a la operación de avanzar el cursor). También en este caso habrá que tener en cuenta los casos especiales.

En cuanto al manejo del espacio que liberamos al hacer una supresión, podremos seguir dos criterios diferentes: administrarlos de forma dinámica o estática.

- Dinámica: habrá que insertar en la lista de los espacios libres el espacio liberado a causa de la supresión en la lista implementada. Esto implica, claro está, programar las acciones necesarias para insertar en la lista de los espacios libres; en consecuencia, se trata de un método dinámico no automático.
- Estática permanente: no se inserta nunca, en la lista de los espacios libres, el espacio liberado a causa de la supresión en la lista implementada.
- Estática periódica o temporaria: habrá que realizar una recorrida, periódicamente, de los arreglos para detectar cuáles elementos han quedado liberados e incorporarlos (insertarlos) a la lista de los espacios libres.

### CON UN SOLO ARREGLO

Este caso es posible cuando puede emplearse el tipo registro (`struct` en C, `record` en Pascal).

El arreglo soporte tendrá como tipo base un registro cuyos campos deberán ser las distintas partes de cada uno de los elementos de la lista: VIPD y puntero o punteros (cuando se trate de una lista bidireccional).

La Figura 11.28 muestra un ejemplo de definición de un tipo `blista` en C que sirve como soporte de una lista bidireccional de caracteres.

```
#define N   10;

struct unNodo{
  char vipd;
  int ps;
  int pa;
};

struct blista {
     struct unNodo nodos[N];
     int acc, lib, cur, curaux;
}
```

**Figura 11.28. Definición en C de un tipo `struct blista` que sirve de estructura soporte para una lista bidireccional de caracteres usando como estructura soporte un único arreglo para las VIPDs y los punteros al siguiente y al anterior, más un puntero al primero de la lista (`acc`), un puntero al primero de la lista de espacios libres (`lib`), un cursor (`cur`) y un cursor auxiliar (`curaux`).**

El manejo y administración, tanto de la lista simulada como la de los espacios libres, se realiza de la misma manera que en el caso de varios arreglos, tal como fue explicado en la implementación con arreglos separados y en la sección 11.2 “Administración de los espacios libres”. Obviamente, la referencia a los elementos y al puntero (o a los punteros para las bidireccionales) será en el registro y no en los arreglos.

# 12 RECURSIVIDAD

## 12.1 NOCIONES GENERALES

La *recursión* o *recursividad* es una técnica de resolución de algunos problemas particulares. En general, decimos que la definición de un concepto es recursiva si el concepto es definido en términos de sí mismo. Así, por ejemplo, en matemática, se da el nombre de recursión a la técnica consistente en definir una función en términos de sí misma. Un ejemplo bastante conocido de definición de una *función recursiva* es el de la función factorial (!), definida para los enteros positivos:

$! : \mathbb{N}_0 \rightarrow \mathbb{N}$

$$n! = \begin{cases} n \times (n-1)! & \text{si } n > 0 \\ 1 & \text{si } n = 0 \end{cases}$$

Existen múltiples ejemplos de definiciones recursivas. Otro ejemplo es la siguiente definición del conjunto de los números naturales ($\mathbb{N}$):

a) 1 pertenece a $\mathbb{N}$.

b) Si $n$ pertenece a $\mathbb{N}$, entonces $n+1$ pertenece a $\mathbb{N}$.

En toda definición recursiva distinguimos dos *partes*:

- *Caso base o elemental.* Existe uno o más casos (generalmente de menor tamaño) que pueden resolverse sin recursión. Es decir, para uno o más casos se conoce la definición del concepto mismo, es decir, no se necesita de la recursión para definirlo. Por ejemplo, en las funciones definidas recursivamente, para uno o más valores del domino se conoce el valor que corresponde al rango (o codomino). Así, en el ejemplo de la función factorial el caso base es 0! = 1.
- *Una definición recursiva, circular o caso general.* Decimos que la definición es circular porque se hace 'en términos de sí misma', es decir, incorpora una o más aplicaciones de sí misma dentro de la definición. Por ejemplo, en la función factorial, la definición circular corresponde a :

$$n! = n \times (n-1)!, \text{ si } n > 0$$

Así, $n!$ queda definido en términos de $(n-1)!$, el cual a su vez está definido en términos de $(n-2)!$ y así sucesivamente hasta llegar a 0!, que es el caso base o valor del dominio conocido, en donde la recursión para.

En Computación, la recursión se encuentra presente en la definición de módulos (funciones, procedimientos, subrutinas, etc.) recursivos así como en la definición de tipos de datos recursivos.

La mayor parte de los lenguajes de programación soportan la recursión permitiendo que un módulo pueda invocarse a sí mismo. Por medio de la recursión, podemos de realizar repeticiones. Ya hemos visto cómo las repeticiones en los lenguajes de programación imperativos, y en particular en el caso de C, las implementamos por medio de sentencias de iteración. Veremos que también pueden hacerse empleando recursión.

## 12.2 MÓDULOS RECURSIVOS

En general, decimos que un *módulo es recursivo* cuando el módulo (función, procedimiento, subrutina, etc.) se encuentra definido en términos de sí mismo. Es decir, un modulo es recursivo si (a) dentro de su cuerpo existe una invocación a sí mismo, lo que correspondería al caso recursivo o general; (b) existe uno o varios casos elementales que pueden resolverse directamente sin necesidad de recursión, que correspondería al o los casos bases.

Entonces, ¿por qué escribir módulos recursivos, cuando existen las iteraciones? En la mayoría de los casos, una función recursiva se puede reemplazar por una función iterativa; sin embargo, muchas veces la solución recursiva resulta más clara e intuitiva que la iterativa; además, permite en algunos casos, definir módulos complejos de manera muy compacta. Por otro lado, hay que tener en cuenta que no debe recomendarse de forma indiscriminada el uso de la recursión. El programador debe hacer un balance entre los beneficios de un módulo sencillo, escrito con mayor facilidad, contra el tiempo adicional de ejecución así como la posibilidad de dificultades para encontrar y corregir errores inherentes a una solución recursiva.

Como toda repetición, la recursión también puede ser finita o infinita, dependiendo de la condición que la controla. La condición que controla la recursión es el casos base (o los casos base, si hay más de uno), que permite(n) que la recursión "termine" o "pare".

En la Figura 12.1 damos una definición de una función en C que calcula la factorial de un número entero `n`. En la rama del falso existe, en el valor retornado por la función, una invocación recursiva `(factorial(n – 1))`. Esta es la invocación que hace que `factorial` sea una función definida en forma circular. El caso base es `n` igual a `0`. Véase en el ejemplo que para `n == 0` la función devuelve `1`. La función `factorial` está, así, definida recursivamente: tiene una definición circular y al menos un caso base o valor del dominio para el cual se conoce el valor del rango que retorna la función.

```
long factorial(int n){
 if (n == 0)
   return 1;
 else
   return n * factorial(n - 1);
}
```

**Figura 12.1. Ejemplo de función recursiva en C.**

Supongamos, ahora, que dicha función es invocada en un programa de la siguiente manera:

```
printf("%ld \n", factorial(2));
```

Puede verse en la Figura 12.2 que para `factorial(2)` se realizan tres llamadas de la función, antes que se obtenga un valor para `factorial(0)`(caso base) y puedan entonces calcularse los valores correspondientes a las llamadas anteriores que han quedado pendientes de resolver.

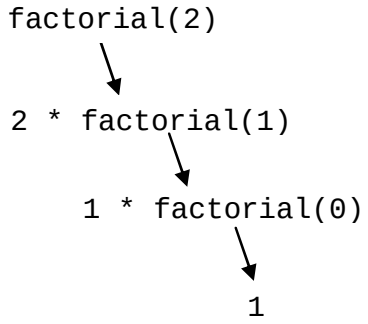


**Figura 12.2. Cálculo de `factorial(2)`.**

En efecto, en la primera llamada, siendo `n` igual a `2` (y en consecuencia mayor a cero), se invoca nuevamente a `factorial` con el parámetro `n–1`, es decir, `factorial(1)`; en el segundo llamado, siendo `n` igual a `1` (y en consecuencia mayor a cero) se invoca nuevamente a `factorial` con el valor `n – 1` (cero), y es entonces, en

el tercer llamado, que siendo `n` igual a `0`, la función `factorial` retorna `1`, pudiendo ahora calcularse, yendo hacia atrás, los distintos valores de las expresiones pendientes no resueltas.

Este mecanismo, que el compilador se encarga de implementar, implica dejar pendiente el cálculo de la función hasta tanto se obtenga un valor conocido para la función. En el caso de la función `factorial` esto ocurre cuando `n` es igual a `0`, donde `factorial(0)` devuelve el valor `1`.

## 12.3 RECURSIVIDAD. EJEMPLOS

Veamos algunos otros ejemplos de definiciones recursivas que contemplen otros aspectos de la recursión tales como el hecho de que en la definición pueda haber más de una invocación recursiva, tal como sucede con la definición de la función que calcula el número de Fibonacci.

### SUCESIÓN DE FIBONACCI

Fibonacci, o más correctamente Leonardo de Pisa o Leonardo Pisano, nació en Pisa, Italia, alrededor de 1175. Hijo de Guglielmo Bonacci, comenzó –quizás– a usar el nombre Fibonacci como una abreviatura de "filius Bonacci", hijo de Bonacci. Debido a que su padre tenía contactos comerciales con el norte de África, Fibonacci creció con una educación influenciada por la cultura de los moros, especialmente su matemática. Estos a su vez habían sido influenciados por la cultura hindú que utilizaban la notación posicional decimal (empleada hoy en día) con los símbolos arábigos, especialmente el cero. Fibonacci introdujo esta matemática, incluida la forma, o algoritmos, de realizar las operaciones elementales, en la Europa de esa época, que todavía utilizaba el sistema romano de numeración.

En su libro "Liber abbaci" (Libro del Ábaco o Libro del Cálculo), no solo introdujo el sistema hindú–arábigo de numeración y cálculo, sino también el problema de los conejos como una forma de los lectores del libro de practicar las operaciones introducidas. Este problema puede resumirse así:

"Si una pareja de conejos son puestos en un campo, y si los conejos toman un mes en madurar, y si luego producen una nueva pareja de conejos cada mes a partir de ese momento, ¿cuántas parejas de conejos habrá en doce meses?"

Se supone que los conejos no mueren ni se escapan. Este problema dio origen a la sucesión de números: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Así los términos de la sucesión pueden verse como:

| *término* | $f_0$ | $f_1$ | $f_2$ | $f_3$ | $f_4$ | $f_5$ | $f_6$ | $f_7$ | $f_8$ | $f_9$ | $f_{10}$ | $f_{11}$ | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *valor* | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | ... |

**Figura 12. 3. Sucesión de Fibonacci**

El matemático francés Edouard Lucas (1842-1891), le dio el nombre de números de Fibonacci a esta sucesión, además de descubrir importantes propiedades de la misma.

Cada término de la sucesión se calcula sumando los dos anteriores, salvo los dos primeros, que son 0 y 1 respectivamente. Es decir, cualquier término $f_n$ de esta sucesión puede calcularse por medio de una definición recursiva:

| | |
|---|---|
| $f_0 = 0$ | El primer término de la sucesión es el 0 (caso base). |
| $f_1 = 1$ | El segundo término de la sucesión es el 1 (caso base). |
| $f_n = f_{n-1} + f_{n-2}$ | El tercer término y los sucesivos se calculan recursivamente. |

Obsérvese que esta definición recursiva, en la parte de la definición circular o caso general, la función queda definida en término de dos aplicaciones recursivas de la función, en lugar de una como ocurría, por ejemplo, con la definición de la función factorial. La definición cuenta, además, con dos casos base o elementales: el valor de la función para el primer y segundo término de la sucesión.

La Figura 12. 4 muestra la definición en C de la función `fibonacci` para calcular un término de la sucesión de Fibonacci teniendo en cuenta la definición recursiva dada más arriba. La definición de la función tiene dos condiciones de parada de la recursión: una para el valor 0 y otra para 1, que se encuentran englobados en una única condición (`n < 2`), como se ve en la Figura 12. 4. Hay que hacer notar que, por razones de simplicidad y para mantener coherencia con la definición dada antes, la función del ejemplo no controla que el parámetro `n`, correspondiente al término de la serie que se quiere calcular, sea mayor o igual a 0. Será la función que invoque a la función `fibonacci` la que deberá controlarlo.

```
int fibonacci(int n){
  if (n < 2) return n;
  else return fibonacci(n – 1) + fibonacci(n – 2);
}
```

**Figura 12. 4. Una solución recursiva en C para la sucesión de Fibonacci.**

Para calcular `fibonacci(n)`, se debe calcular el valor de `fibonacci(n-1)` y el de `fibonacci(n-2)`. Ambas invocaciones, a su vez, tendrán que realizar el cálculo de `fibonacci((n-1)-1)` + `fibonacci((n-1)-2)`, y de `fibonacci((n-2)-1)` + `fibonacci((n-2)-2)`, y así sucesivamente, en cada caso. Estas invocaciones pueden verse representadas como una estructura de árbol. En la Figura 12. 5 se ve el árbol que surge de las invocaciones a la función `fibonacci` para `n` igual a 5. Notar que el árbol se representa "al revés", con la "raíz" arriba y las "ramas" hacia abajo. Las hojas del árbol son los valores del dominio conocido o casos base, `n` = 1 y `n` = 2, que devuelven 0 y 1 respectivamente.

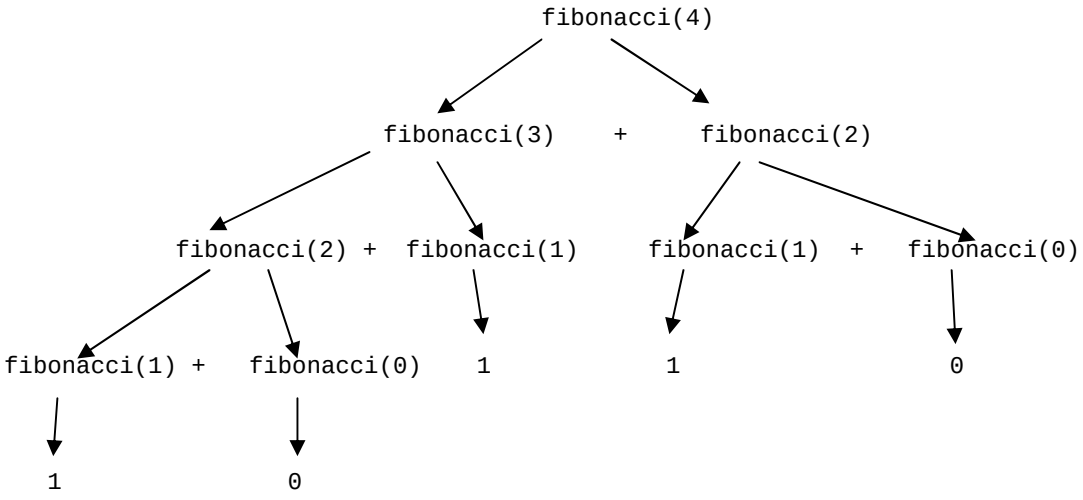


**Figura 12. 5. Árbol de invocaciones para el cálculo del quinto término de la sucesión de Fibonacci.**

## "TORRES DE HANOI"

Un ejemplo de solución recursiva de un problema es el del juego llamado *Torres de Hanoi*, inventado por el matemático francés Edouard Lucas y que se vendía como un juego ya a fines del siglo XIX. Aparentemente, Lucas se inspiró en una leyenda oriental que decía que un grupo de monjes debían mover una pila de sesenta y cuatro

discos de oro de un poste a otro. De acuerdo con la leyenda, cuando terminasen su tarea tanto el templo como el mundo se terminarían también. El juego consiste en pasar una torre o pila de objetos de un lugar a otro, respetando las siguientes reglas:

(a) Sólo puede emplearse un lugar auxiliar, (además del original de partida y el del destino);

(b) Sólo puede moverse un objeto por vez;

(c) No puede nunca apilarse un objeto encima de otro más pequeño.

Generalmente, se representa tal como el juego se vende: los objetos son discos con un agujero en el medio y tres clavijas montadas cada una sobre una base, que representan las pilas (la de partida u origen, la de llegada o destino y la auxiliar), como se ve en la Figura 12.6(a), donde hay tres discos en la clavija de origen. Nosotros podemos representarlo como tres pilas de números naturales como se ve en la Figura 12.6 (b). Notemos que el tamaño de los discos está en relación directa con los números naturales que los representan en las pilas de la Figura 12.6(b).

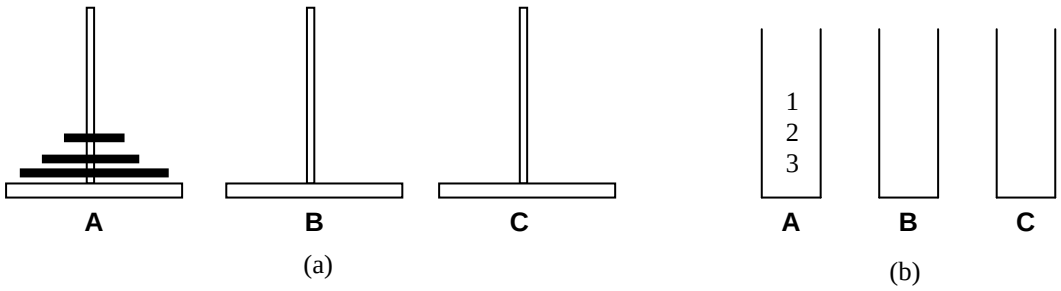


**Figura 12.6. Representación del juego de las Torres de Hanoi**

Una forma de encarar este problema es suponer el caso más simple: cuando se trata de pasar un solo número esto es fácil: se pasa y terminamos, ya que no queda nada por pasar ni en la pila de origen (**A**), ni en la auxiliar (**B**).

Compliquemos un poco más el caso a tratar y supongamos que hay dos números a pasar, entonces: 1° pasamos el que está en el tope de **A** (pila de origen) a **B** (pila auxiliar), luego el nuevo tope de **A** a **C** (pila de destino), y por fin el que estaba en **B** a **C** y terminamos ya que no queda nada por pasar ni en la pila de origen (**A**), ni en la auxiliar (**B**). Estos pasos pueden verse en la Figura 12.7. Esto nos permite deducir que para pasar dos elementos son necesarios tres movimientos.

| **Paso** | **Pilas** | | | **Mov.** |
|---|---|---|---|---|
| | **A** (origen) | **B** (auxiliar) | **C** (destino) | |
| inicio | 1<br>2 | | | |
| 1 | 2 | 1 | | **A** → **B** |
| 2 | | 1 | 2 | **A** → **C** |
| 3 | | | 1<br>2 | **B** → **C** |

**Figura 12.7. Movimientos para Torre (pila) con dos discos.**

Si hay tres números a pasar, podemos ver la sucesión de pasos en la Figura 12. 8. Vemos que la solución para tres elementos consiste en aplicar la solución para dos elementos, antes de pasar el tercer elemento. En primer lugar, se pasa del origen (**A**) a la auxiliar (**B**) todos los elementos menos uno. Esto sucede en los primeros tres pasos. Hay que notar que para ello se emplea A como origen y B como destino y C como auxiliar. Luego, en el paso 4, el último elemento que queda en el origen (**A**) se pasa al destino (**C**). Por último se pasan los elementos que están en la pila **B** a la pila destino **C** (pasos 5 a 7). Para estos pasos, se utiliza la pila **A** como auxiliar, la **B** como origen y **C** como destino.

En la Figura 12.9 puede verse la función en C que implementa la solución recursiva. Se supone que las declaraciones previas del tipo `stack_of_int` y de las otras funciones allí empleadas para manipular las pilas (`copy`, `insert`, `suppress`, etc.) han sido dadas y pueden ser empleadas. Además de las tres pilas empleamos un parámetro extra (`n`) de tipo `int` para indicar la cantidad de discos a pasar desde la pila de origen a la de destino.

| Paso | Pilas | | | Mov. |
|---|---|---|---|---|
| | **A** (origen) | **B** (auxiliar) | **C** (destino) | |
| inicio | 1<br>2<br>3 | | | |
| 1 | 2<br>3 | | 1 | **A → C** |
| 2 | 3 | 2 | 1 | **A → B** |
| 3 | 3 | 1<br>2 | | **C → B** |
| 4 | | 1<br>2 | 3 | **A → C** |
| 5 | 1 | 2 | 3 | **B → A** |
| 6 | 1 | | 2<br>3 | **B → C** |
| 7 | | | 1<br>2<br>3 | **A → C** |

**Figura 12. 8. Movimientos para torre (pila) con tres discos.**

```
void hanoi(int n, stack_of_int orig , stack_of_int dst , stack_of_int aux)
{
  int disco;
    if (n == 1){
      disco = copy(orig);
      suppress(orig);
      insert(dst, disco);
    }
    else {
      hanoi(n - 1, orig, aux, dst);   /* pasa n-1 discos de orig a aux*/
                                      /* usando dst como auxiliar */
      hanoi(1, orig, dst, aux);       /* pasa 1 disco de orig a dst */
      hanoi(n - 1, aux, dst, orig);   /* pasa n-1 discos de aux a dst*/
                                      /* usando orig como auxiliar */

   }
}
```

**Figura 12.9. Solución recursiva en C para el problema de las torres de Hanoi.**

Puede verse que la solución consiste en mover primero todos los elementos, menos uno, de la pila original (`orig`) a la auxiliar (`aux`), cualquiera sea la cantidad de elementos, esta es la primera sentencia de invocación recursiva de la función `hanoi`:

```
hanoi(n – 1, orig, aux, dst);
```

Luego, cuando queda un solo elemento, se mueve a la pila de destino (`dst`), esta es la segunda sentencia de invocación de la función `hanoi`:

```
hanoi(1, orig, dst, aux);
```

Finalmente, se mueven todos los elementos de la pila auxiliar a la de destino, que es la tercera sentencia de invocación de la función `hanoi`:

```
hanoi(n – 1, aux, dst, orig);
```

Está claro que, cada una de estas sentencias de invocación de la función `hanoi`, implican, a su vez, el llamado de ellas mismas en cada una. Salvo la segunda sentencia de invocación recursiva ya que como su primer parámetro es `1`, y ese es el punto de parada, allí termina la recursión: es el caso de mover un solo elemento.

## 12.4 TIPOS DE RECURSIVIDAD

Podemos hablar de distintos tipos de recursividad, dependiendo de la forma en que ocurran las llamadas recursivas. Así, por un lado, podemos hablar de *recursividad lineal* vs. *recursividad múltiple*. Otra forma de clasificación habla de *recursividad directa* vs. *recursividad indirecta*.

### RECURSIVIDAD LINEAL Y RECURSIVIDAD MÚLTIPLE

En la *recursividad lineal*, dentro del módulo recursivo existe una única invocación a sí mismo, como por ejemplo en el caso de la función `factorial` (ver Figura 12.1). En general, los módulos recursivos lineales son fácilmente transformados a su versión iterativa.

Por el contrario, en la *recursividad múltiple*, el módulo se invoca a sí mismo más de una vez dentro de una misma activación. Este tipo de módulos son más difíciles de llevar a su forma iterativa. Dos ejemplos de funciones recursivas múltiples son la función que implementa la sucesión de Fibonacci y la de las Torres de Hanoi (Figura 12. 4 y Figura 12.9, respectivamente).

Como una variante de la recursión múltiple, podemos tener *recursión anidada*, que ocurre cuando dentro de una invocación recursiva ocurre como parámetro otra invocación recursiva. Un ejemplo de esto puede verse en la función de la Figura 12.10, que implementa la función de Ackerman.

```
int ackerman(int n, int m)
{
    if (n == 0) return m + 1;
    else if(m == 0) return ackerman(n - 1, 1);
    return ackerman(n - 1, ackerman(n, m - 1));
}
```

**Figura 12.10. Función de Ackerman.**

### RECURSIVIDAD DIRECTA E INDIRECTA

Hasta ahora hemos visto sólo lo que llamamos *recursividad directa*: un módulo que se llama a sí mismo, como en los ejemplos anteriores.

Existe también la *recursividad indirecta* que es aquella que se produce cuando se tienen varios módulos que se llaman unos a otros formando un ciclo. Al igual que en la recursividad directa, es necesario contar con al menos un punto de parada de la recursión. Por ejemplo, un módulo *A* llama a otro *B*, y este a su vez llama *A*, y así sucesivamente como, por ejemplo, en el programa de la Figura 12.11 que llama a dos funciones mutuamente recursivas que determinan si un número natural *n* es par o si *n* es impar, de acuerdo a lo siguiente:

- *n* es par si *n* − 1 es impar,
- *n* es impar si *n* − 1 es par,
- 0 es par

```
#include <stdio.h>

int par(int);
int impar(int);

main(){

   printf("Ingrese un numero natural:");
   scanf("%d",&n);
   if (par(n))
      printf("El numero %d es par\n", n);
   else
     printf("El numero %d es impar\n", n);
}

int par(int n)
{
  if (n == 0) return 1;
  return impar(n-1);
}

int impar(int n)
{
  if (n == 0) return 0;
  return par(n-1);
}
```

**Figura 12.11. Recursividad indirecta.**

## 12.5 PROFUNDIDAD DE LA RECURSIÓN

Las invocaciones recursivas de módulos traen aparejadas, en la mayor parte de los lenguajes de programación, un uso bastante importante de memoria. Esto es así ya que cada invocación a un módulo, sea éste recursivo o no, implica una 'suspensión' del flujo normal de ejecución del módulo invocante para 'saltar' a ejecutar las sentencias del módulo invocado. Este 'salto' implica, en algún punto, un retorno a la sentencia siguiente a la de invocación del módulo, para continuar así con la ejecución interrumpida. Este retorno debe ser hecho de manera ordenada, es decir que, por ejemplo, las variables deben conservar los valores que tenían antes de la invocación. Para esto generalmente se emplea una pila o stack, a la que se la denomina usualmente *pila de ejecución* o *stack de ejecución* en donde se va guardando toda la información del estado de ejecución de cada módulo invocado. A este conjunto de datos, entre los que se encuentra, por ejemplo, la dirección de retorno (el lugar del programa donde se encuentra la sentencia siguiente a la invocación), las variables locales y los parámetros del módulo, se lo llama *registro de activación* del módulo.

Cada vez que se invoca un módulo se coloca en el tope del stack de ejecución su correspondiente registro de activación, el cual es suprimido del tope al alcanzar el fin del módulo y retornar el control al módulo invocante.

Tales activaciones de los módulos pueden anidarse hasta cualquier nivel (en particular en forma recursiva) mientras lo permita la capacidad del stack de ejecución y no ocurra un desborde (overflow). Todo ello nos lleva al concepto de *profundidad de la recursión* que es la cantidad máxima de registros de activación del módulo recursivo que se van a encontrar simultáneamente en el stack de ejecución para una entrada de datos dada.

Cuando la recursión es lineal, como por ejemplo en el caso de la función `factorial`, la profundidad de la recursión coincide con la cantidad de veces que la función es invocada para un conjunto de argumentos dados. Esto es así sólo para el caso en que haya una sola invocación en el módulo recursivo (recursión lineal). Así, por ejemplo, para `n` igual a `2` la función `factorial` (Figura 12.1 ) es invocada `3` veces, es decir diremos que la profundidad es `3`, mientras que para `n` igual a `10000` la profundidad será `10001`, lo que implica, en algún

punto de la ejecución, 10001 registros de activación al mismo tiempo en el stack de ejecución!

Es por esto que la profundidad debe ser razonable y finita. Finita porque es condición que el programa termine y razonable porque de lo contrario la recursión puede hacerse bastante lenta y tomar un tiempo excesivamente largo en terminar; peor aún, puede llegar a desbordar la pila de ejecución. Para ello véase el ejemplo de la Figura 12.10, que implementa la función de Ackerman, y analice la profundidad para `n=2` y `m=3`; ello permitirá darse una idea de cómo la recursividad puede ocultar la cantidad de veces que una función puede ejecutarse.

La profundidad tiene directa relación con lo que se llama el 'costo' de un programa. Este costo tiene que ver con dos aspectos de los programas: la cantidad de memoria que el programa y sus datos ocupan y el tiempo de ejecución del mismo. Así podemos medir el tiempo de ejecución de una función recursiva por la cantidad de veces que la misma se ejecuta. Por otro lado, la cantidad de memoria que la función ocupa podemos relacionarla con el uso que la misma haga del stack de ejecución, es decir la cantidad máxima de registros de activación, que en una invocación dada, haya simultáneamente en el stack de ejecución.

## 12.6 DATOS RECURSIVOS

Las listas de elementos (los cuales pueden o no ser a su vez, listas) es un ejemplo de datos que pueden definirse recursivamente. Así, dados *e*, que denota un elemento de lista y ∅ que denota la lista vacía, podemos definir una lista *L*, en forma recursiva, diciendo que:

$$L = \varnothing \quad \vee \qquad L = e\,L$$

Otro ejemplo es un registro que se referencia a sí mismo por medio de un campo de tipo puntero que apunta a un registro del mismo tipo. La definición dada en Figura 12.12, es un ejemplo en C de este tipo de datos recursivos, donde se define un tipo `struct nodo`, en donde cada registro de este tipo cuenta con un campo para la `vipd` y otro campo, `next`, para el puntero al siguiente elemento, el que es, a su vez, un nuevo nodo, por lo que el tipo de `next` es puntero a `struct nodo`. En el campo `next` vemos la recursividad: `next` tomará como valor o bien un puntero a un nuevo elemento de la estructura de datos (el cual es de tipo `struct nodo)` o el valor `NULL`, que en este caso sería el caso base o punto de parada de la recursión.

```
struct nodo {
    char vipd;
    struct nodo *next;
}
```

**Figura 12.12. Definición en C de un tipo de dato recursivo.**

Este tipo de estructuras recursivas, que se autoreferencian, pueden usarse para soportar estructuras de datos tales como listas, colas, pilas, árboles, etc. Así, por ejemplo, la definición dada en la Figura 12.12 podríamos usarla para implementar una lista unidireccional, donde cada elemento de la lista contendrá por un lado la `vipd` y por otro el puntero al próximo (`next`), el que tomará el valor `NULL` cuando se trate del último elemento de la lista. Podemos, así, declarar una variable `lisChar` de tipo puntero a `struct nodo` para mantener una lista de caracteres, la cual inicialmente se encontrará vacía, es decir, apuntará a `NULL` de la siguiente manera:

```
struct nodo *lisChar = NULL;
```

A efectos de claridad y economía de escritura en el ejemplo, definiremos un sinónimo para el tipo `struct nodo *`:

```
typedef struct nodo *PNodo;
```

Luego la declaración de `lisChar`, puede también ser hecha como

```
PNodo lisChar = NULL;
```

`lisChar` no es otra cosa que el acceso a la lista. Así, cuando `lisChar` tenga elementos, en lugar de apuntar a `NULL`, apuntará al primer elemento de la lista (de tipo `struct nodo`). En estas listas así definidas, no necesariamente sus elementos (nodos) se encuentran contiguos. Lógicamente, sin embargo, los nodos 'parecen' estar contiguos, como en toda lista. En la Figura 12.13(a) vemos una representación gráfica de la lista vacía, y en (b) una instancia posible de la lista, después que se han insertado cuatro caracteres:

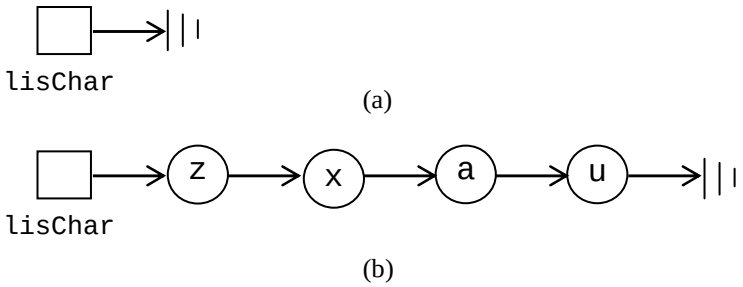


**Figura 12.13. Una lista vacía (a) y con los caracteres `z, x, a, u` (b).**

Viendo la definición de los datos dados en la Figura 12.12, resulta más natural tratar una lista así definida de manera recursiva, aunque nada impide que sea tratada de manera iterativa. Así, por ejemplo, si quisiéramos recorrer la lista imprimiendo sus elementos, una solución recursiva podría ser la dada en la Figura 12.14 mientras que una solución iterativa se da en la Figura 12.15; en ambas se supone la definición de la Figura 12.12 y la definición del sinónimo `PNodo` para `struct nodo *`. Si las comparamos, la solución recursiva resulta más natural para este tipo de dato recursivo.

```
void imprimeLista(PNodo lis)
{
  if (lis != NULL){
       printf("%c ",(*lis).vipd);
       imprimeLista((*lis).next);
   }
}
```

**Figura 12.14. Función recursiva que recorre una lista de caracteres**

```
void imprimeLista(PNodo lis)
{
  while (lis != NULL){
       printf("%c ",(*lis).vipd);
       lis = (*lis).next;
   }
}
```

**Figura 12.15. Función iterativa que recorre una lista de caracteres**

Si suponemos, además, que `lischar` es la lista de la Figura 12.13(b) que se encuentra en memoria para ser recorrida, el resultado de la invocación `imprimeLista(lischar)` en ambos casos será: `z x a u`.

¿Qué pasaría si en la definición de la Figura 12.14 se intercambiase el orden de las sentencias que se encuentran dentro del bloque del `if`, es decir, si fuese

```
imprimeLista((*lis).next);
printf("%c ",(*lis).vipd);?
```

La función imprimiría la lista en orden inverso, es decir, `u a x z`. ¿Por qué ocurre esto? Porque, en el primer caso, primero se ejecuta la sentencia de invocación a `printf` y luego la invocación recursiva; mientras que en el segundo caso primero se ejecuta la invocación recursiva y hasta que no se llegue al final de la lista, que es el caso base o punto de parada de la recursión (`lis == NULL`), el control no volverá al `imprimeLista` de la llamada

anterior que quedó pendiente, en donde se deberá ejecutar la sentencia siguiente que, en este caso, corresponde al `printf`. Al ejecutarse el `printf`, se mostrará el valor al cual se encuentra apuntando `lis` en esa activación de `imprimeLista`, que en este caso será el último elemento de `lischar`. Es entonces que termina la ejecución de la penúltima invocación a `imprimeLista` y retorna el control a la activación anterior, en donde se ejecutará el `printf` mostrando el penúltimo elemento y así sucesivamente hasta alcanzar la primera activación de `imprimeLista`, que quedó pendiente, en donde se imprimirá el primero.

Sin embargo si intercambiamos las sentencias que se encuentran dentro del bloque del `while` de la Figura 12.15, esto no cambia el orden de impresión. ¿Por qué? Además, esto produciría un error. ¿Cuál?

La definición de lista de la Figura 12.12 es simple y no contempla todas las operaciones que hemos visto tienen las listas. Más adelante se da una definición más completa y más apta para soportar las operaciones indicadas.

## 12.7 IMPLEMENTACIÓN DE LISTAS CON DATOS RECURSIVOS

Las listas son estructuras de datos que pueden definirse de forma recursiva. Hemos visto en las implementaciones previas (ver Sección 11.5) que para las listas, necesitamos de cosas tales como cursores y un acceso. Una posible definición recursiva de una lista unidireccional de caracteres, que contemple todos estos aspectos, podría ser la que se muestra en la Figura 12.16.

```
struct nodo{
    char vipd;
    struct nodo *next;
};

typedef struct nodo Nodo;

typedef struct {
    Nodo *acceso;     /* acceso a la lista*/
    Nodo *cur;        /* cursor de la lista */
    Nodo *curaux;     /* cursor auxiliar de la lista*/
} ulist_of_char;
```

**Figura 12.16. Tipo `ulist_of_char` que soporta una lista unidireccional de caracteres con sus cursores.**

No será necesario mantener un apuntador a los espacios libres ya que al usar asignación dinámica de la memoria a través de tipos punteros, será el compilador quién se encargue de la administración de los espacios libres y no nosotros.

Veamos, entonces, cómo serían las funciones que dan soporte a las operaciones típicas sobre una lista implementada con datos recursivos como los de la definición dada en la Figura 12.16.

*Inicialmente* (función `create`), la lista se encuentra vacía y, en consecuencia, el `acceso` será igual a `NULL`, mientras que los cursores (`cur` y `curaux`), que se encontrarán fuera de la estructura, también se encontrarán apuntando a `NULL`.

La condición de *lista vacía* (función `isEmpty`) es simple: si el acceso (`acceso`) es igual a `NULL` entonces no hay nada en la lista.

La condición de *lista llena* o mejor dicho de cuándo no hay más lugar en la estructura soporte para seguir insertando (función `isFull`) es cosa ahora de quien administra los espacios libres: el compilador C a través de la asignación y desasignación dinámica de la memoria, como vimos en la sección anterior. Es decir, deberemos interrogar al compilador para saber si tiene especio libre para poder hacer una inserción. Esto lo haremos al pedir

espacio por medio de la llamada a la función `malloc`.

¿Cuándo el cursor está *fuera de la estructura* (función `isOos`)? Esto sucede cuando el cursor (`cur`) es igual a `NULL`.

¿Cómo colocamos el cursor en la primera posición de la lista (función `reset`)? Bastará con hacer apuntar al cursor `cur` al mismo lugar que apunte el acceso `acceso` de la lista; esto significa que si la lista se encuentra vacía, el cursor quedará apuntando a `NULL`.

¿Cómo se hace *avanzar* el cursor al siguiente elemento (función `forwards`)? Esto se hace asignando al cursor el valor del puntero al siguiente del elemento apuntado por el cursor. Recordemos que también hay que mover el cursor auxiliar `curaux`, el que siempre viene una posición atrás de `cur`.

Para *copiar* el elemento corriente, es decir, el elemento apuntado por el cursor (función `copy`) lo único que tendremos que hacer es devolver la `vipd` del nodo al cual apunta el cursor `cur`.

Para *insertar* (función `insert`) hay que conseguir un lugar libre, pidiéndoselo al administrador de los espacios libres: el compilador C, siendo que éste es el encargado a través del tipo puntero que estamos empleando. Esto lo hacemos usando la función `malloc` y `sizeof`. Como el nuevo elemento se inserta entre dos elementos existentes o en el primer lugar (que equivale a insertarlo antes del primer elemento) o después del último, hay que controlar de qué caso se trata.

Si es entre dos elementos habrá que hacer que el elemento anterior (el que es apuntado por el cursor auxiliar `curaux`) apunte al nuevo y el nuevo elemento apunte al corriente (que está siendo apuntado por el cursor, `cur`), y luego hacer que el cursor apunte al nuevo. Si es en la primera posición, habrá que hacer que el acceso de la lista (`acceso`) apunte al nuevo elemento y el nuevo elemento apunte al corriente (que está siendo apuntado por el cursor, `cur`), y luego hacer que el cursor apunte al nuevo elemento.

Para *suprimir* (función `suppress`) hay que distinguir entre dos casos: supresión del primer elemento de la lista o de cualquier otro. Si se trata de cualquier otro, entonces hay que hacer que el puntero al siguiente (`next`) del elemento anterior al apuntado por el cursor (que está siendo apuntado por el cursor auxiliar `curaux`), apunte al elemento apuntado por el puntero al siguiente (`next`) del elemento apuntado por el cursor (`cur`). Luego, hacer que el cursor (`cur`) apunte al elemento apuntado por el puntero al siguiente (`next`) del elemento al que `cur` apunta (esto se parece bastante a la operación de avanzar el cursor). Asimismo, habrá que liberar los bytes ocupados por el nodo suprimido haciendo uso de la función `free`. Si se trata de suprimir el primer elemento, se deberá proceder de forma similar, excepto que en lugar de usar puntero al elemento anterior (`curaux`), que no existe en este caso, haremos referencia al acceso de la lista `(acceso).`

## 12.8 CASOS PARTICULARES DE RECURSIÓN

Al emplear recursión se plantean algunos problemas que no pueden resolverse solamente con las definiciones de recursión como las que hemos visto hasta ahora. Muchas veces, las soluciones a estos problemas son provistas por estructuras propias de algunos lenguajes. En otros casos deben ser resueltos, si es posible, con las estructuras existentes del lenguaje empleado.

Es importante, sin embargo, poder reconocerlos para poder plantear su solución. Veamos algunos de estos casos. Es de hacer notar que los ejemplos dados para ilustrar los distintos casos, a menos que se diga lo contrario, hacen uso de la definición dada en el ejemplo de la Figura 12.16.

### INICIALIZACIÓN DE LA RECURSIÓN

Muchas veces al programar y cuando debe implementarse una repetición, es necesario realizar tareas previas una única vez. Por ejemplo, la inicialización de variables con algunos valores especiales iniciales, los cuales, en algunos casos, puede ser necesario actualizar en cada repetición. En la mayor parte de los lenguajes de programación es imposible realizar esto dentro del módulo recursivo, ya que estas acciones deben realizarse una sola vez, antes de empezar la recursión. Si estas acciones se encontrasen dentro del módulo recursivo se ejecutarían en cada una de las invocaciones recursivas al módulo.

A menos que el lenguaje empleado provea algún mecanismo de inicialización previa a la recursión, esto puede resolverse de varias maneras:

Poniendo, dentro del módulo recursivo, una condición que sólo tenga efecto, sea verdadera, por ejemplo, en la primera invocación y falsa en las subsiguientes. La condición controla la ejecución de las acciones que deben ser hechas una sola vez antes de que se realice efectivamente la recursión. Esta solución es bastante pobre ya que implica que dicha condición será evaluada cada vez que el módulo sea invocado en cada ejecución de la recursión. Además el módulo deberá invocarse con un parámetro extra que permita evaluar la condición.

Otra solución podría ser la de emplear variables globales, que serán inicializadas antes de la primera invocación al módulo recursivo, por afuera de este. Como ya sabemos, el uso de variables globales está desaconsejado, en consecuencia esta no es una buena solución ya que implica que el módulo tiene efectos colaterales que pueden pasar desapercibidos.

Una tercera solución consiste en declarar un módulo que “envuelva” al módulo recursivo, si el lenguaje de programación lo permite, de manera tal que nadie pueda invocar al módulo recursivo sin haber hecho las correspondientes inicializaciones; es decir, declarando el módulo recursivo dentro de otro módulo, que es donde se realizan las tareas previas que sean necesarias y el primer llamado al módulo recursivo. Por supuesto que este módulo “envolvente” no será recursivo.

Esta última solución parece la mejor, sin embargo, dado que en C no se pueden anidar las definiciones de las funciones, deberemos encontrar una solución alternativa. La misma consiste en realizar las inicializaciones previas en el módulo invocante al recursivo o en otro módulo el cual siempre deberá ser invocado antes de la primera invocación al módulo recursivo. Notemos que esta solución no impide que algún desprevenido invoque al módulo recursivo sin antes haber invocado o hecho las correspondientes inicializaciones.

En la Figura 12.17, se muestra un ejemplo donde se imprimen los elementos de una lista de caracteres. Un criterio para realizar esta operación ‘imprimir lista’ implica la impresión completa de la lista; para esto es necesario asegurarse que el cursor se encuentra al principio de la lista, es decir que antes de invocar la función habrá que incializarlo haciendo `reset`. Si esto se hiciera dentro de la función, la operación de `reset` se ejecutaría en cada invocación de la función recursiva. Para el caso, no queda otra solución que confiar que quien invoque al módulo lo haga ejecutando previamente la correspondiente inicialización del cursor por medio de un llamado a la función `reset`.

```
void imprimeLista(ulist_of_char lis)
{
      if (!isOos(lis)){
         printf("elemento: %c\n", copy(lis));
         forwards(&lis);
         imprimeLista(lis);
   }
}
```

**Figura 12.17. Función recursiva que imprime los elementos de una lista.**

Por ejemplo, asumiendo que la variable `miLista` ha sido declarada de tipo `list_of_char`, la siguiente invocación, previo `reset` del cursor, imprimirá los elementos de `miLista`:

```
reset(&miLista);
imprimeLista(miLista);
```

Cabe notar que el cursor de `miLista`, después de imprimir la lista, quedará apuntando a la primera posición; esto es así porque estamos pasando `miLista` por valor a la función `imprimeLista`.

## ACTUALIZACIÓN DE VALORES ENTRE LLAMADAS

Algunas soluciones recursivas implican la necesidad de mantener, de una invocación a otra, e inclusive de pasar de una llamada a otra un valor; por ejemplo, imprimir un elemento de una lista y su posición en la lista implicará pasar entre las distintas invocaciones la posición a imprimir.

Algunas soluciones posibles para este caso son:

Pasar uno o más parámetros, de acuerdo con las necesidades. Esto implica que el módulo deberá invocarse la primera vez con el o los parámetros con algún valor inicial razonable. Por ejemplo, en el caso anterior implicaría en la primera invocación pasar un 0 o un 1.

Si, por el contrario, no se desea que el usuario del módulo pase estos valores como parámetros ya que estos son siempre los mismos (por ejemplo siempre hay que empezar de cero o uno, etc.) entonces, una solución para esto es "envolver" al módulo recursivo, declarándolo dentro de otro módulo que es el que da a dichos parámetros sus valores iniciales e invoca al módulo recursivo. Por supuesto que el módulo que "envuelve" al recursivo no es recursivo. Esta solución, ya vimos que no nos servirá en el caso del lenguaje C.

Usar una o más variables globales, de acuerdo con las necesidades, es otra solución, aunque no una buena ya que implica que el módulo tendrá efectos colaterales que pueden pasar desapercibidos.

En la Figura 12.18 se muestra un ejemplo donde se imprimen los elementos y la posición de los elementos de una lista de caracteres. Para esto, fue necesario pasar como parámetro de la función recursiva no sólo la lista sino también una variable para llevar la posición del elemento corriente.

```
void imprimeLista(ulist_of_char lis, int pos)
{
      if (!isOos(lis)){
         printf("elemento %d: %c\n", pos, copy(lis));
         forwards(&lis);
         imprimeLista(lis, pos+1);
      }
}
```

**Figura 12.18. Función recursiva que imprime la posición y los elementos dentro de una lista.**

Así, la primera vez que se invoque a `imprimeLista`, el parámetro `pos` deberá tomar el valor `1`. Por ejemplo, asumiendo que la variable `miLista` ha sido declarada de tipo `ulist_of_char`, la siguiente invocación, previo `reset` del cursor de `miLista`, imprimirá la posición y todos los elementos de `miLista`:

```
reset(&miLista);
imprimeLista(miLista, 1);
```

## REPETICIONES ANIDADAS

Cuando el problema a resolver requiere que la solución tenga repeticiones anidadas, ya sea una o más, este anidamiento puede reflejarse en la solución, ya sea iterativa o recursiva.

Un ejemplo de esto puede verse en la función `suma` de la Figura 12.19 la cual devuelve la suma los elementos de una matriz de números reales. Este es un problema en el que, al resolverlo de forma iterativa, usamos dos

iteraciones anidadas; sin embargo, en la versión recursiva lo hemos podido resolver con un único módulo recursivo. La función toma tres parámetros: la matriz, y el número de fila y columna de la posición de la matriz que se está sumando. Inicialmente, `i` y `j` tomarán valor 0 ambos y a medida que la ejecución avance se irá incrementando `j` hasta que alcance la última posición. Cuando esto ocurre, la invocación recursiva cambia, incrementándose el valor de `i` en 1 y volviendo `j` a 0. De esta manera, la matriz se irá recorriendo por filas. Trate de escribir una versión de esta función en la cual la matriz es recorrida por columnas.

```
#include <stdio.h>
#define MAXFIL 2
#define MAXCOL 4

float suma(float [][MAXCOL], int, int);

main(){
    float a[MAXFIL][MAXCOL] = {1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0};
    printf("La suma de los elementos de a es:%f", suma(a, 0, 0));
    return 0;
}

/* Función recursiva que suma los elementos de una matriz*/
float suma(float x[][MAXCOL], int i, int j){
    if (i < MAXFIL)
          if (j < MAXCOL) return x[i][j]+ suma(x, i, j+1);
          else return suma(x, i+1, 0);
    else return 0;
}
```

**Figura 12.19. Programa que define función recursiva suma que devuelve la suma de los elementos de una matriz.**

En el ejemplo de la Figura 12.20 se puede ver otro ejemplo de un problema que involucra repeticiones anidadas. Se trata de la función `potencia`, que eleva un número natural `n` a un exponente entero positivo `m`, usando sólo la operación de suma. La función recursiva `potencia` calcula $n^m$ como $n \times n^{m-1}$, donde $n^{m-1}$ a su vez es calculado como $n \times n^{m-2}$ y así sucesivamente hasta que el exponente se hace `0`, que es cuando se alcanza el caso base y la función retorna `1`. A su vez, cada producto es calculado llamando a la función recursiva `producto`, la cual a su vez realiza el producto por sumas sucesivas usando la recursión en lugar de una sentencia de repetición. Las dos iteraciones anidadas han sido reemplazadas por dos funciones recursivas.

```
/* Función recursiva que calcula la potencia de n elevado al exponente m */
int potencia(int n, int m)
{
    if (m == 0) return 1;
    else return producto(n, potencia(n, m-1));
}

/* Función recursiva que calcula el producto de n por m */
int producto(int n, int m)
{
    if (m == 0) return 0;
    else return n + producto(n, m-1);
}
```

**Figura 12.20. Solución recursiva para calcular la potencia de dos naturales por sumas sucesivas**

## 12.9 ITERACIÓN Y RECURSIÓN

Todos los problemas que implican algún tipo de repetición aceptan una solución iterativa pero también pueden

ser solucionados en forma recursiva. Veamos, con mayor detalle, ambas aproximaciones a la repetición y comparémoslas para así poder decidir frente a un problema dado qué tipo de solución será más conveniente de usar.

Tanto las estructuras de iteración como la recursión involucran repetición. La iteración lo hace de forma explícita por medio de las estructuras de repetición que proveen los lenguajes, tales como por ejemplo el `for` o el `while` en el C; la recursión lo hace de forma implícita por medio de repetidas invocaciones a la función.

Ambas involucran condiciones de parada, la iteración cuando la condición que controla la misma alcanza un determinado valor de verdad y la recursión cuando se alcanza un caso base; en ambos casos puede decirse que la condición es la misma o muy parecida, ya que en ambos casos el valor de verdad se dará cuando se alcance el caso base. Asimismo, tanto la iteración como la recursión pueden ocurrir de forma infinita; en la iteración cuando la condición de parada nunca alcance el correspondiente valor de verdad y en la recursión cuando los pasos recursivos no reduzcan el problema de manera tal que converja al caso base o caso más simple.

La recursión tiene sus aspectos negativos. Implica un gasto extra tanto de tiempo como de espacio, debido a las repetidas invocaciones de la función recursiva: cada llamado crea una copia en el stack de ejecución del ambiente de la función consumiendo memoria y tiempo. Entonces, ¿por qué elegir la recursión? La elección de una solución recursiva o iterativa dependerá de varios factores tales como las características particulares del problema así como de los medios que se cuenten para materializar la solución. En la próxima sección damos alguno de los pros y contras de cada una.

## 12.10 VENTAJAS E INCONVENIENTES DE LA RECURSIÓN

La recursividad tiene sus propias ventajas e inconvenientes. Entre las primeras podemos citar el hecho que nos permite definir un conjunto potencialmente infinito de objetos por medio de una expresión finita. Esto ocurre por ejemplo, cuando definimos datos por medio de expresiones recursivas.

Los algoritmos o soluciones recursivas son útiles, particularmente, cuando existe una definición recursiva, ya que nos permite crear soluciones compactas desde el punto de vista del texto del programa y, muchas veces, adaptadas directa y fácilmente a partir de la definición del problema. En particular, por ejemplo, cuando los datos a tratar se encuentran definidos de forma recursiva: listas definidas con datos recursivos, árboles, etc. Otro ejemplo muy claro es el de los parsers descendentes recursivos que se construyen sobre la base de la definición recursiva de la gramática de un lenguaje.

Sin embargo, una ejecución recursiva puede rápidamente “irse de las manos”. Pensemos sino en la solución recursiva implementada en la Figura 12. 4 para el cálculo de la sucesión de Fibonacci, donde cada invocación realiza a su vez otras dos invocaciones. Así, por ejemplo, para calcular el 30º término serán necesarias 1.664.079 invocaciones a la función. Esto es, seguramente, un desafío para el poder computacional de muchas computadoras.

Por otro lado, la recursividad oculta, y algunas veces en forma muy efectiva, ya no sólo el número de veces que algo se ejecuta sino también la existencia de repeticiones en el programa. En ese sentido, la recursividad no contribuye a hacer equivalentes las estructuras estática y dinámica de un programa, sino todo lo contrario.

Es cierto que, como sucede en la mayor parte de los casos, la recursividad se materializa en máquinas que no son en sí mismas recursivas y que a lo sumo permiten realizarla con mayor o menor facilidad.

# APÉNDICE A LÓGICA

## A.1 INTRODUCCIÓN

La Lógica es una disciplina que tiene una larga tradición. Muchos autores consideran a Aristóteles (384 a.c. - 322 a.c.) como el creador, o como uno de los primeros pensadores, que sistemáticamente realizó una exposición y uso formal de una metodología de razonamiento que puede entenderse que es propia del ámbito de la Lógica. Posteriormente, especialmente desde el siglo XIX, con George Boole (1815–1864), la Lógica se formaliza y adquiere herramientas algebraicas que le permiten manipular símbolos y conceptos. El Diccionario de la Real Academia la define:

**lógica.** (Del lat. *logĭca*, y este del gr. λογική). f. Ciencia que expone las leyes, modos y formas del conocimiento científico. || **2.** Tratado de esta ciencia. *Escribió una lógica que fue muy comentada*. || ~ **borrosa,** o ~ **difusa.** f. La que admite una cierta incertidumbre entre la verdad o falsedad de sus proposiciones, a semejanza del raciocinio humano. || ~ **formal,** o ~ **matemática.** f. La que opera utilizando un lenguaje simbólico artificial y haciendo abstracción de los contenidos. || ~ **natural.** f. Disposición natural para discurrir con acierto sin el auxilio de la ciencia. || ~ **parda.** f. coloq. **gramática parda.**

Aunque hay muchas otras definiciones y explicaciones sobre qué es la Lógica (quizás tantas como autores hay que se dedican a ella), podemos decir, por lo menos en una introducción como esta, que la Lógica es la ciencia o la disciplina que estudia la forma de exponer *argumentos*. ¿Qué es un argumento? Según la Real Academia un argumento es:

**argumento.** (Del lat. *argumentum*). m. Razonamiento que se emplea para probar o demostrar una proposición, o bien para convencer a alguien de aquello que se afirma o se niega.

Hay muchas 'lógicas', o quizás mejor dicho, muchas ramas de la Lógica. Nosotros nos limitaremos en éste capítulo a dar una breve introducción a una de las formas más sencillas de ella: la llamada Lógica Proposicional.

Esta Lógica, por emplear un lenguaje formal muy cercano al lenguaje de la Matemática, juega un papel fundamental en la Ciencia de la Computación y por ende en Programación.

Es por ello que antes de comenzar a ver qué es programar es necesario tener una idea básica de Lógica Proposicional. Aquellos familiarizados con el tema pueden saltar este capítulo.

## A.2 PROPOSICIONES

Llamaremos *proposición* (o *enunciado*, o *enunciado proposicional*) a toda expresión del lenguaje para la cual tenga sentido *uno y sólo uno* de los valores siguientes:

Verdadero o Falso

Es decir, que se pueda afirmar sobre dicha expresión que es verdadera o falsa, pero no ambas. La Lógica Proposicional es una lógica bivalente, porque acepta sólo dos valores: Verdadero y Falso. Hay lógicas multivalentes (o multivaluadas) que aceptan tres o más valores.

Por ejemplo, son proposiciones:

Si se añade ácido sulfúrico al agua, se obtiene una camioneta.

2 + 2 = 4

3 + 2 = 4

3 es múltiplo de 8

Todos los estudiantes de computación están preocupados por el hambre y el desempleo que hay en nuestro país.

En cambio no son proposiciones, las siguientes:

¿Qué distancia hay de la Tierra a la Luna?

¡Qué frío!

La Cátedra de programación.

$3 + x = 16$

$x - 3y + 6 = 0$

Las expresiones f) a j) no son proposiciones porque no es posible afirmar si ellas son verdaderas o falsas.

Sin embargo las expresiones i) y j) pueden llegar a transformarse en *enunciados proposicionales* si se le asignan valores a las variables *x* e *y*. Sólo haciendo ese reemplazo podremos afirmar que esas expresiones son verdaderas o falsas y que por lo tanto son proposiciones.

A *x* e *y* se les denomina *variables de objeto*, y las expresiones que las contienen se llaman *funciones proposicionales*. Es decir, que las expresiones i) y j) son funciones proposicionales que, cuando se les da valores a las variables *x* e *y*, se transforman en proposiciones.

Para referirnos a un enunciado cualquiera utilizaremos las letras p, q, r, s, t, etc. Las llamaremos *variables de enunciados* o *variables proposicionales*. Estas variables proposicionales, entonces, pueden representar cualquier proposición que pueda tomar el valor Verdadero o Falso. Se llama *valor de verdad* de una proposición al valor –Verdadero o Falso– que la misma tiene. Así, por ejemplo, el valor de verdad de la proposición $3 > 4$ es Falso.

Está claro que si se habla, por ejemplo, de la variable proposicional p, ella no es de por sí ni verdadera ni falsa. Lo será recién cuando se le dé una interpretación, o sea cuando se reemplace p por una proposición concreta, aunque sí podemos afirmar que, por ser p una variable proposicional, ella puede tomar uno de los dos valores: Verdadero o Falso.

Por ejemplo, si p es reemplazada por la frase "todos los gatos son negros", su valor será Falso, pero si a p le asignamos el significado de "el cigarrillo es perjudicial para la salud", su valor será verdadero ya que, luego de estudios realizados recientemente, se habría descubierto una relación química –no estadística– entre la combustión del tabaco y el cáncer de pulmón.

Sin embargo, las manipulaciones y aseveraciones que veremos más adelante en éste capítulo, son independientes de la "verdad" o "falsedad" de las proposiciones involucradas. Dichas manipulaciones y aseveraciones son útiles por la forma misma que tienen, independientemente del significado que se asigne a ellas, por eso es que podemos reemplazar proposiciones concretas por símbolos (las letras p, q, r, s, t, etc., en nuestro caso) y las manipulaciones y aseveraciones que hagamos con ellas seguirán siendo equivalentes a las que podamos hacer con proposiciones concretas.

Muchos lógicos distinguen entre *validez* y *valor de verdad*. La validez queda reservada para los argumentos (o razonamientos), mientras que el valor de verdad es para las proposiciones. Siendo que en esta introducción sólo trataremos con proposiciones no veremos nada sobre validez.

Emplearemos como símbolos para Verdadero una **V** y para Falso una **F**. Por supuesto que dos símbolos cualesquiera pueden emplearse para ello, aunque, obviamente, una vez elegidos no pueden intercambiarse. Así, también, se emplean **True** (en Inglés) para Verdadero y **False** (en Inglés) para Falso, o **T** y **F** respectivamente. Muchas veces se emplean también los dígitos **1** y **0**. Cualquiera de ellos pueden emplearse como sinónimos: **Verdadero**, **True**, **T**, **V**, **1**, y por otro lado: **False**, **F**, **0**.

## A.3 OPERADORES, CONECTIVAS O FUNTORES

Las proposiciones vistas hasta ahora son *proposiciones simples*. Las *proposiciones complejas o compuestas* son aquellas que se forman empleando dos proposiciones que se encuentran relacionadas entre sí por un *operador*, *funtor* o *conectiva lógica*. Cada una de estas proposiciones puede ser, a su vez simple o compuesta.

En el lenguaje común hay ciertas *partículas* gramaticales que se usan para conectar dos enunciados, dando así origen a un nuevo enunciado, compuesto de los otros dos, más la partícula gramatical que los une. Lo mismo se puede hacer con las proposiciones en la Lógica.

El valor de verdad de este nuevo enunciado lógico depende del (o también podemos decir *es función del*) valor de verdad de los enunciados que se unen junto con el significado que tiene la partícula que los une.

Estas partículas se denominan, en Lógica, *funtores* o *funtores de verdad*. También se los conoce como *operadores lógicos* o *conectivas lógicas*.

Podemos, entonces, decir que si

p , q

son enunciados o proposiciones, entonces también lo son:

p **y** q p **o** q

El valor de verdad de estas proposiciones compuestas depende del valor de p y de q como así también del sentido o significado que le demos a **y** y a **o**. En el lenguaje natural (por ejemplo el castellano) la interpretación de una expresión lingüística no siempre es única, puede ser ambigua. En la Matemática (la que incluye la Lógica que estamos exponiendo), por el contrario, el lenguaje empleado es no ambiguo (a menos que haya un error), es decir que cada expresión tiene una única interpretación.

Las proposiciones compuestas también se las llaman *expresiones lógicas*. Seguiremos usando las letras p, q, r, s, t, etc. para denotar proposiciones simples o compuestas de aquí en más. De resultar necesario distinguir entre proposiciones simples o compuestas, utilizaremos las minúsculas (p, q, r, s, t, etc.) para las simples y mayúsculas (P, Q, R, S, T, etc.) para las expresiones lógicas o proposiciones compuestas. También emplearemos paréntesis: ( ), cuando sea necesario para evitar ambigüedades en los casos de proposiciones compuestas.

Las conectivas lógicas relacionan dos (no más ni menos), proposiciones entre sí. Asumiendo, como dijimos, que estas proposiciones pueden tener dos valores posibles (verdadero o falso), y si, como dijimos, la relación de dos proposiciones por medio de una conectiva puede tener uno de dos valores posibles (verdadero o falso), las posibles conectivas no pueden ser más de dieciséis.

Sin embargo, no todas estas combinaciones son interesantes. En realidad, utilizando manipulaciones algebraicas, puede demostrarse que con solo una de ellas pueden representarse todas las otras (hay dos que permiten esto). Sin embargo esto sería poco práctico en el uso corriente, por lo que en general se consideran unas cuantas más, que son las que trataremos a continuación.

| p |
|---|
| **V** |
| **F** |

Figura A.1

El significado de las conectivas se establece a través de lo que se llama *tabla de verdad* para la conectiva dada. Una tabla de verdad es una lista completa de todos los valores de verdad posibles de una proposición (simple o compuesta). Así, por ejemplo, la tabla de verdad de una proposición p es la que se muestra en la Figura A.1.

Entonces, por medio de una tabla se establece qué valores (verdadero o falso) puede tomar la nueva proposición (compuesta de dos proposiciones más la conectiva), teniendo en cuenta qué valores tienen cada una de las proposiciones que la conforman. Por supuesto, es necesario que todos los casos posibles de valores de verdad para

cada una de las proposiciones queden establecidos.

De ahora en más denotaremos con **V** el valor *verdadero* de una proposición y con **F** el valor *falso*.

Si en Lógica las expresiones tienen una única interpretación o significado (pueden tomar un solo valor: verdadero o falso), daremos, entonces, un único significado a las **conectivas** (partículas) **y** y **o**.

## CONJUNCIÓN

Llamaremos *conjunción* a la conectiva **y**. Emplearemos el símbolo $\wedge$ para denotarla. Esta conectiva se conoce en inglés como **and**.

La tabla de verdad para la conjunción es la que se muestra en la Figura A.2.

| p | q | p ∧ q |
|---|---|---|
| **V** | **V** | **V** |
| **V** | **F** | **F** |
| **F** | **V** | **F** |
| **F** | **F** | **F** |

Figura A.2.

O sea que $p \wedge q$ es *verdadero* (**V**) solamente cuando *ambas* p y q son verdaderas. En otras palabras, cuando la proposición p es *verdadera* y la proposición q es *verdadera*. La proposición $p \wedge q$ será *falsa* (**F**) en todos los otros casos: ya sea que p o q sean ambas falsas o que una cualquiera lo sea.

Ejemplos:

1) $(2 < 3) \wedge (13 < 20)$ es *verdadero* (**V**) pues $(2 < 3)$ es *verdadero* (**V**) y $(13 < 20)$ también es *verdadero* (**V**).

2) $(2 < 3) \wedge (6 \times 2 = 20)$ es *falso* (**F**) pues si bien $(2 < 3)$ es *verdadero* (**V**), $(6 \times 2 = 20)$ es *falso* (**F**).

## DISYUNCIÓN

La conectiva **o** se la conoce como *disyunción*. Esta conectiva, tiene, sin embargo, en el lenguaje natural, dos interpretaciones: una *incluyente* y otra *excluyente*, empleándose en ambos casos el mismo símbolo (**o**), lo que se presta, en más de una ocasión a ambigüedades de interpretación.

Esta ambigüedad no es posible en la Lógica. Esto nos obliga a tener dos símbolos para cada caso, y en consecuencia dos Tablas de Verdad distintas.

Emplearemos el símbolo $\vee$ para denotar la conectiva **o** *incluyente*, con la tabla de verdad que se ve en la Figura A.3. En Inglés se la conoce como **or**. Decimos, entonces, que $p \vee q$ es *verdadero* (**V**) cuando ya sea p o q es *verdadero* o ambas lo son.

| p | q | p ∨ q |
|---|---|---|
| **V** | **V** | **V** |
| **V** | **F** | **V** |
| **F** | **V** | **V** |
| **F** | **F** | **F** |

Figura A.3.

Ejemplos:

1) $(2 < 3) \vee (13 < 20)$ es *verdadero* (**V**) pues $(2 < 3)$ es *verdadero* (**V**) y $(13 < 20)$ también es *verdadero* (**V**).

2) $(2 < 3) \vee (6 \times 2 = 20)$ es *verdadero* (**V**) pues $(2 < 3)$ es *verdadero* (**V**), y no importa que $(6 \times 2 = 20)$ es *falso* (**F**). Véase la diferencia entre este ejemplo y el mismo pero para la conectiva $\wedge$.

El símbolo generalmente empleado para la conectiva **o** *excluyente* es $\veebar$. Sin embargo los símbolos $\neq$ ; $\oplus$ son también usados. La tabla de verdad para ella es la que se muestra en la Figura A.4. Vemos que $p \veebar q$ es *verdadero* (**V**) sólo cuando o p es *verdadero* o q es *verdadero*, no cuando ambas lo son.

| p | q | $p \veebar q$ |
|---|---|---|
| **V** | **V** | **F** |
| **V** | **F** | **V** |
| **F** | **V** | **V** |
| **F** | **F** | **F** |

Figura A.4.

Ejemplos:

1) $(2 < 3) \veebar (13 < 20)$ es *falso* (**F**) pues $(2 < 3)$ es *verdadero* (**V**) y $(13 < 20)$ también es *verdadero* (**V**).Véase la diferencia entre este ejemplo y el mismo pero para la conectiva $\vee$.

2) $(2 < 3) \veebar (6 \times 2 = 20)$ es *verdadero* (**V**) pues $(2 < 3)$ es *verdadero* (**V**), y $(6 \times 2 = 20)$ es *falso* (**F**).

Es interesante comparar ambas Tablas para detectar la diferencia entre ellas.

En sentido *incluyente* p **o** q $(p \vee q)$ es *verdadero* (**V**) si ambas proposiciones son verdaderas o si una sola de ellas lo es. Si ambas son falsas, entonces $p \vee q$ será *falso* (**F**).

En sentido *excluyente* p **o** q $(p \veebar q)$ es *verdadero* (**V**) si una sola de ellas lo es. Si ambas son falsas, o si ambas son verdaderas, entonces $p \veebar q$ será *falso* (**F**).

## CONDICIONAL O IMPLICACIÓN

Otro funtores o conectiva interesante es la llamada *condicional* o *implicación*. El símbolo empleado comúnmente para representar esta conectiva es $\Rightarrow$. La tabla de verdad es la que muestra la Figura A.5.

| p | q | $p \Rightarrow q$ |
|---|---|---|
| **V** | **V** | **V** |
| **V** | **F** | **F** |
| **F** | **V** | **V** |
| **F** | **F** | **V** |

Figura A.5.

A la primera proposición (p) se la denomina *antecedente*. A la segunda proposición (q) se la denomina *consecuente*. El *condicional* es *falso* (**F**) solo cuando el antecedente es *verdadero* (**V**) y el consecuente es *falso* (**F**). En todos los otros casos el condicional es *verdadero* (**V**). Especial atención merece el hecho que si $p \Rightarrow q$ es *verdadero* (**V**) no implica nada sobre los valores de verdad de p y de q, estos pueden ser verdaderos o falsos. Sin embargo, si el antecedente es verdadero la implicación será verdadera si el consecuente es también verdadero. Esto debe interpretarse como que de una verdad no puede "deducirse" una falsedad.

Ejemplos:

1) $(2 < 3) \Rightarrow (13 < 20)$ es *verdadero* (**V**) pues $(2 < 3)$ es *verdadero* (**V**) y $(13 < 20)$ también es *verdadero* (**V**).

2) $(2 < 3) \Rightarrow (6 \times 2 = 20)$ es *falso* (**F**) pues $(2 < 3)$ es *verdadero* (**V**), y $(6 \times 2 = 20)$ es *falso* (**F**).

Hasta ahora las conectivas mostradas tienen una tabla de verdad que es deducible "intuitivamente": la conjunción es verdadera solo si ambas proposiciones lo son, la disyunción es verdadera cuando alguna de las dos proposiciones es verdadera o ambas lo son. Sin embargo, la tabla de verdad de la implicación no es deducible tan intuitivamente como las otras dos. La implicación puede expresarse de varias maneras en español, por ejemplo:

(a) "si p entonces q",

(b) "p implica q",

(c) "p es condición suficiente de q",

(d) "q es condición necesaria de p",

(e) "p solo si q",

(f) "q si p".

Así, cuando la implicación tiene el valor de verdad *verdadero* podemos afirmar que: "*Si* p *es verdadero entonces* q *es verdadero*", pero esta es una afirmación *condicional*. En efecto no afirmamos que siempre que p es verdadero q también lo es. Decimos, afirmamos, con el condicional, que *cuando* la implicación tiene el valor de verdad verdadero, SI p es verdadero ENTONCES q también es verdadero. Esto se abrevia muchas veces diciendo "*Si* p *entonces* q".

Es interesante la expresión (e) más arriba que nos dice que si la implicación es verdadera, p es verdadera sólo si q también es verdadera. Importantes son también las expresiones © y (d) que nos muestran con precisión cuáles son los casos de condiciones necesarias y suficientes.

### CONDICIÓN SUFICIENTE Y CONDICIÓN NECESARIA

Dados p y q, decimos que p es una *condición suficiente* de q cuando la verdad de p garantiza la verdad de q. Por el contrario, q es *condición necesaria* de p cuando la falsedad de q garantiza la falsedad de p.

Diremos entonces que, en la implicación, el antecedente es condición suficiente del consecuente y que el consecuente es condición necesaria del antecedente. Entonces "p solo si q" asevera que q es condición necesaria de p. Entonces podemos decir que en la implicación, la condición necesaria del antecedente es el consecuente de dicha implicación.

### BICONDICIONAL O EQUIVALENCIA

Otro funtor o conectiva, es la llamada *bicondicional* o *equivalencia*. El símbolo empleado comúnmente para representar esta conectiva es $\Leftrightarrow$. También se emplea el símbolo $\equiv$. La tabla de verdad es la que muestra la Figura A.6.

El bicondicional es *verdadero* (**V**) solamente (*si y solo si*) ambas proposiciones son verdaderas o falsas, al mismo tiempo.

Ejemplos:

1) $(2 < 3) \Leftrightarrow (13 < 20)$ es *verdadero* (**V**) pues $(2 < 3)$ es *verdadero* (**V**) y $(13 < 20)$ también es *verdadero* (**V**).

2) $(2 < 3) \Leftrightarrow (6 \times 2 = 20)$ es *falso* (**F**) pues $(2 < 3)$ es *verdadero* (**V**), y $(6 \times 2 = 20)$ es *falso* (**F**).

| p | q | p ⇔ q |
|---|---|---|
| V | V | V |
| V | F | F |
| F | V | F |
| F | F | V |

Figura A.6.

| p | q | p ⇒ q | q ⇒ p | (p ⇒ q) ∧ (q ⇒ p) |
|---|---|---|---|---|
| V | V | V | V | V |
| V | F | F | V | F |
| F | V | V | F | F |
| F | F | V | V | V |

Figura A.7.

Nuevamente, como en el caso de la implicación, nos encontramos con una afirmación condicionada: cuando la equivalencia tiene el valor de verdad *verdadero*, p es *verdadero* SI Y SOLO SI q también es *verdadero*. "*Si y solo si*" se suele abreviar como *sii*. Podemos abreviar todo esto diciendo: "p *sii* q". Algunas veces esta conectiva se expresa también: "*Si y solo si* p es *verdadero entonces* q es *verdadero*".

Es de hacer notar que

$(p \Leftrightarrow q) = (p \Rightarrow q) \wedge (q \Rightarrow p)$

Esto puede comprobarse con la tabla de verdad de la Figura A.7.

### NEGACIÓN

Toda proposición puede ser *negada*, es decir que si podemos *afirmar*, por ejemplo, "está lloviendo", pudiendo ser esta proposición verdadera o falsa, también podemos afirmar "no está lloviendo", y esta proposición también puede ser verdadera o falsa. Decimos que toda proposición puede ser *negada* y hacemos esto anteponiendo a la proposición el operador *no*.

| p | ~ p |
|---|---|
| V | F |
| F | V |

Figura A.8.

Este operador no relaciona dos proposiciones, sino que afecta a una sola proposición. Se podría decir que los funtores vistos hasta ahora (*conjunción*, *disyunción*, *condicional* y *bicondicional*) son binarios porque afectan (relacionan) dos proposiciones, mientras que la negación es unaria, porque solo afecta una proposición. Así, si p es una proposición, *no* p también lo es.

Este operador se denota con el símbolo ~ (muchas veces también se usa el símbolo ¬). La tabla de verdad es la que muestra la Figura A.8.

## A.4 EQUIVALENCIAS LÓGICAS

Llamamos *tautología* a toda expresión compuesta cuyo valor de verdad es siempre *verdadero* cualquiera sea el valor de verdad de las proposiciones que la componen.

Llamamos *contradicción* a toda expresión compuesta que es siempre *falsa* cualquiera sea el valor de verdad de las proposiciones que la componen.

A aquellas expresiones que pueden ser verdaderas o falsas de acuerdo con el valor de verdad de las proposiciones que la componen se las denominan *contingentes*.

Sea P una expresión lógica, si P es una *tautología* entonces ~P es una contradicción y viceversa.

La forma de determinar si una expresión lógica es una tautología o una contradicción es analizando la correspondiente expresión empleando la tabla de verdad.

Tautologías y equivalencias lógicas son muy útiles para operar y simplificar expresiones lógicas complejas.

Sean P, Q, R, expresiones lógicas. Algunas de las tautologías y equivalencias lógicas más comunes son las siguientes:

*Tautologías Básicas*

$P \vee (\sim P)$ $P \Rightarrow P$ $P \Leftrightarrow P$ $(P \wedge Q) \Rightarrow P$

*Identidad*

$P \wedge \mathbf{V} = P$ $P \vee \mathbf{F} = P$

*Doble Negación*

$\sim (\sim P) = P$

*Conmutatividad*

$P \vee Q = Q \vee P$ $P \wedge Q = Q \wedge P$ $P \Leftrightarrow Q = Q \Leftrightarrow P$

*Asociatividad*

$(P \vee Q) \vee R = P \vee (Q \vee R)$ $(P \wedge Q) \wedge R = P \wedge (Q \wedge R)$

$(P \Leftrightarrow Q) \Leftrightarrow R = P \Leftrightarrow (Q \Leftrightarrow R)$

*Distributividad*

$P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$ $P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$

*Leyes de DeMorgan*

$\sim (P \wedge Q) = \sim P \vee \sim Q$ $\sim (P \vee Q) = \sim P \wedge \sim Q$

*Transitividad*

$(P \Rightarrow Q) \wedge (Q \Rightarrow R) \Rightarrow (P \Rightarrow R)$ $(P \Leftrightarrow Q) \wedge (Q \Leftrightarrow R) \Rightarrow (P \Leftrightarrow R)$

## APÉNDICE B TABLA ASCII (AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE)

| Valor decimal | Carácter | Valor decimal | Carácter | Valor decimal | Carácter | Valor decimal | Carácter |
|---|---|---|---|---|---|---|---|
| 0 | NUL | 32 | Espacio (SP) | 64 | @ | 96 | ‘ |
| 1 | SOH | 33 | ! | 65 | A | 97 | a |
| 2 | STX | 34 | ” | 66 | B | 98 | b |
| 3 | ETX | 35 | # | 67 | C | 99 | c |
| 4 | EOT | 36 | $ | 68 | D | 100 | d |
| 5 | ENQ | 37 | % | 69 | E | 101 | e |
| 6 | ACK | 38 | & | 70 | F | 102 | f |
| 7 | BEL | 39 | ’ | 71 | G | 103 | g |
| 8 | BS | 40 | ( | 72 | H | 104 | h |
| 9 | HT | 41 | ) | 73 | I | 105 | i |
| 10 | LF | 42 | * | 74 | J | 106 | j |
| 11 | VT | 43 | + | 75 | K | 107 | k |
| 12 | FF | 44 | , | 76 | L | 108 | l |
| 13 | CR | 45 | - | 77 | M | 109 | m |
| 14 | SO | 46 | . | 78 | N | 110 | n |
| 15 | SI | 47 | / | 79 | O | 111 | o |
| 16 | NUL | 48 | 0 | 80 | P | 112 | p |
| 17 | DC1 | 49 | 1 | 81 | Q | 113 | q |
| 18 | DC2 | 50 | 2 | 82 | R | 114 | r |
| 19 | DC3 | 51 | 3 | 83 | S | 115 | s |
| 20 | DC4 | 52 | 4 | 84 | T | 116 | t |
| 21 | NAK | 52 | 5 | 85 | U | 117 | u |
| 22 | SYN | 54 | 6 | 86 | V | 118 | v |
| 23 | ETB | 55 | 7 | 87 | W | 119 | w |
| 24 | ↑ (CAN) | 56 | 8 | 88 | X | 120 | x |
| 25 | ↓ (EM) | 57 | 9 | 89 | Y | 121 | y |
| 26 | → (SUB) | 58 | : | 90 | Z | 122 | z |
| 27 | ← (ESC) | 59 | ; | 91 | [ | 123 | { |
| 28 | Cursor derecha (FS) | 60 | < | 92 | \ | 124 | \| |
| 29 | Cursor izquierda (GS) | 61 | = | 93 | ] | 125 | } |
| 30 | Cursor arriba (RS) | 62 | > | 94 | ^ | 126 | ~ |
| 31 | Cursor abajo (US) | 63 | ? | 95 | _ | 127 | |